

Henk Sips
Dick Epema
Hai-Xiang Lin (Eds.)

LNCS 5704

Euro-Par 2009 Parallel Processing

15th International Euro-Par Conference
Delft, The Netherlands, August 2009
Proceedings



 Springer



Commenced Publication in 1973

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison

Lancaster University, UK

Takeo Kanade

Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler

University of Surrey, Guildford, UK

Jon M. Kleinberg

Cornell University, Ithaca, NY, USA

Alfred Kobsa

University of California, Irvine, CA, USA

Friedemann Mattern

ETH Zurich, Switzerland

John C. Mitchell

Stanford University, CA, USA

Moni Naor

Weizmann Institute of Science, Rehovot, Israel

Oscar Nierstrasz

University of Bern, Switzerland

C. Pandu Rangan

Indian Institute of Technology, Madras, India

Bernhard Steffen

University of Dortmund, Germany

Madhu Sudan

Microsoft Research, Cambridge, MA, USA

Demetri Terzopoulos

University of California, Los Angeles, CA, USA

Doug Tygar

University of California, Berkeley, CA, USA

Gerhard Weikum

Max-Planck Institute of Computer Science, Saarbruecken, Germany

Henk Sips Dick Epema Hai-Xiang Lin (Eds.)

Euro-Par 2009

Parallel Processing

15th International Euro-Par Conference
Delft, The Netherlands, August 25-28, 2009
Proceedings



Springer

Volume Editors

Henk Sips
Dick Epema
Hai-Xiang Lin
Delft University of Technology
Department of Software Technology
Mekelweg 4, 2628 CD Delft, The Netherlands
E-mail: {h.j.sips, d.h.j.epema, h.x.lin}@tudelft.nl

Library of Congress Control Number: 2009932717

CR Subject Classification (1998): B.2.4, B.6.1, C.1.2, C.1.4, D.1.3, F.1.2, G.4, I.6.8, B.8, C.4

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN 0302-9743
ISBN-10 3-642-03868-9 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-03868-6 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

springer.com

© Springer-Verlag Berlin Heidelberg 2009
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12736720 06/3180 5 4 3 2 1 0

Preface

Euro-Par is an annual series of international conferences dedicated to the promotion and the advancement of all aspects of parallel computing. In Euro-Par, the field of parallel computing is divided into the four broad categories of theory, high performance, cluster and grid, and distributed and mobile computing. These categories are further subdivided into 14 topics that focus on particular areas in parallel computing. The objective of Euro-Par is to provide a forum for promoting the development of parallel computing both as an industrial technique and as an academic discipline, extending the frontier of both the state of the art and the state of the practice. The target audience of Euro-Par consists of researchers in parallel computing in academic departments, government laboratories, and industrial organizations.

Euro-Par 2009 was the 15th conference in the Euro-Par series, and was organized by the Parallel and Distributed Systems Group of Delft University of Technology in Delft, The Netherlands. The previous Euro-Par conferences took place in Stockholm, Lyon, Passau, Southampton, Toulouse, Munich, Manchester, Paderborn, Klagenfurt, Pisa, Lisbon, Dresden, Rennes, and Las Palmas de Gran Canaria. Next year, the conference will be held in Sorrento, Italy. More information on the Euro-Par conference series and organization is available on its website at <http://www.europar.org>.

Compared to Euro-Par 2008, the number of topics in the conference remained the same, but there were two changes in the topics. First, the subject of Cloud Computing was added to topic 6, which is now called Grid, Cluster, and Cloud Computing. Secondly, the topic Distributed and High-Performance Multimedia, which did not attract many paper submissions over the last few years, was replaced by the topic Multicore and Manycore Programming. This topic drew the largest number of submissions (31) of all the topics this year.

The reviewing of the papers in each topic was organized and supervised by a committee of at least four persons: a Global Chair, a Local Chair, and two Vice-Chairs. Certain topics with a high number of submissions were managed by a larger committee with more Vice-Chairs. The final decisions on the acceptance or rejection of the submitted papers were made in a meeting of the conference Co-chairs and the Local Chairs of the topics.

The call for papers attracted a total of 256 submissions, representing 41 countries (based on the corresponding authors' countries). A total number of 947 review reports were collected, which makes for an average of 3.7 reviews per paper. In total, 85 papers representing 22 countries were selected for presentation at the conference and inclusion in the conference proceedings, which means an acceptance rate of 33%. Four papers were selected as distinguished papers, which happen to be nicely distributed across the four categories of the conference. These distinguished papers, which were presented in a separate session, are:

1. Alexandru Iosup, *POGGI: Puzzle-Based Online Games on Grid Infrastructures*
2. Giorgos Georgiadis and Marina Papatriantafilou, *A Least-Resistance Path in Reasoning About Unstructured Overlay Networks*
3. Diego Rossinelli, Michael Bergdorf, Babak Hejazialhosseini, and Petros Koumoutsakos, *Wavelet-Based Adaptive Solvers on Multi-core Architectures for the Simulation of Complex Systems*
4. Abhinav Bhatelé, Eric Bohm, and Laxmikant V. Kalé, *A Case Study of Communication Optimizations on 3D Mesh Interconnects*

Euro-Par 2009 was very happy to be able to present three invited speakers of international reputation, who discussed important developments in very different areas of parallel and distributed computing:

1. Michael Perrone (IBM T.J. Watson Research Center, Yorktown Heights, NY, USA): *Multicore Programming Challenges*
2. Henri Bal (Vrije Universiteit, Amsterdam, The Netherlands): *Ibis: A Programming System for Real-World Distributed Computing*
3. Antony Rowstron (Microsoft Research, Cambridge, UK): *What Is in a Namespace?*

Euro-Par 2009 had a record number of 9 workshops co-located with the conference. These workshops had their own Program Committees and managed their own programs. The proceedings of most of these workshops will be published in a joint volume also by Springer. The workshops held in conjunction with Euro-Par 2009 were the:

1. CoreGRID ERCIM Working Group Workshop on Grids, P2P and Service Computing
2. 6th International Workshop on Grid Economics and Business Models (Gecon 2009)
3. 7th International Workshop on Algorithms, Models, and Tools for Parallel Computing on Heterogeneous Platforms (HeteroPar'2009)
4. Third Workshop on Highly Parallel Processing on a Chip (HPPC 2009)
5. Second Workshop on Productivity and Performance (PROPER 2009)
6. Second International Workshop on Real-Time Online Interactive Applications on the Grid (ROIA 2009)
7. UNICORE Summit 2009
8. 4th Workshop on Virtualization in High-Performance Cloud Computing (VHPC'09)
9. XtreemOS Summit

Organizing and hosting the 15th Euro-Par conference in Delft was only possible with the help of many people and organizations. First of all, we thank the authors of all the submitted papers, the members of the topic committees, and all the reviewers for their contributions to the success of the conference. In addition, we thank the three invited speakers for accepting our invitation to present their views on their very interesting areas of expertise in parallel and distributed

computing. We are also grateful to the members of the Euro-Par Steering Committee for their support, and in particular to Luc Bougé for all his advice on the paper submission and reviewing process. We acknowledge the help we obtained from Tomàs Margalef of the organization of Euro-Par 2008. A number of institutional and industrial sponsors contributed toward the organization of the conference. Their names and logos appear on the Euro-Par 2009 website at <http://europar2009.ewi.tudelft.nl/>.

It was our pleasure and honor to organize and host Euro-Par 2009 at Delft University of Technology. We hope all the participants enjoyed the technical program and the social events organized during the conference.

August 2009

Henk Sips
Dick Epema
Hai-Xiang Lin

Organization

Euro-Par Steering Committee

Chair

Chris Lengauer University of Passau, Germany

Vice-Chair

Luc Bougé ENS Cachan, France

European Representatives

José Cunha	New University of Lisbon, Portugal
Marco Danelutto	University of Pisa, Italy
Rainer Feldmann	University of Paderborn, Germany
Christos Kaklamanis	Computer Technology Institute, Greece
Paul Kelly	Imperial College, UK
Harald Kosch	University of Passau, Germany
Thomas Ludwig	University of Heidelberg, Germany
Emilio Luque	Universitat Autònoma of Barcelona, Spain
Tomàs Margalef	Universitat Autònoma of Barcelona, Spain
Wolfgang Nagel	Dresden University of Technology, Germany
Rizos Sakellariou	University of Manchester, UK

Honorary Members

Ron Perrott	Queen's University Belfast, UK
Karl Dieter Reinartz	University of Erlangen-Nuremberg, Germany

Observers

Henk Sips	Delft University of Technology, The Netherlands
Domenico Talia	University of Calabria, Italy

Euro-Par 2009 Organization

Conference Co-chairs

Henk Sips	Delft University of Technology, The Netherlands
Dick Epema	Delft University of Technology, The Netherlands
Hai-Xiang Lin	Delft University of Technology, The Netherlands

Local Organization Committee

Joke Ammerlaan
Pien Rijnink
Esther van Seters
Laura Zondervan

Web and Technical Support

Stephen van der Laan

Euro-Par 2009 Program Committee

Topic 1: Support Tools and Environments

Global Chair

Felix Wolf	Forschungszentrum Jülich, Germany
------------	-----------------------------------

Local Chair

Andy D. Pimentel	University of Amsterdam, The Netherlands
------------------	--

Vice-Chairs

Luiz DeRose	Cray Inc., USA
Soonhoi Ha	Seoul National University, Korea
Thilo Kielmann	Vrije Universiteit, The Netherlands
Anna Morajko	Universitat Autònoma de Barcelona, Spain

Topic 2: Performance Prediction and Evaluation

Global Chair

Thomas Fahringer	University of Innsbruck, Austria
------------------	----------------------------------

Local Chair

Alexandru Iosup	Delft University of Technology, The Netherlands
-----------------	--

Vice-Chairs

Marian Bubak

AGH University of Science and Technology,
Poland

Matei Ripeanu

University of British Columbia, Canada

Xian-He Sun

Illinois Institute of Technology, USA

Hong-Linh Truong

Vienna University of Technology, Austria

Topic 3: Scheduling and Load Balancing**Global Chair**

Emmanuel Jeannot

INRIA, France

Local Chair

Ramin Yahyapour

Technische Universität Dortmund, Germany

Vice-Chairs

Daniel Grosu

Wayne State University, USA

Helen Karatza

University of Thessaloniki, Greece

Topic 4: High-Performance Architectures and Compilers**Global Chair**

Pedro C. Diniz

INESC-ID, Portugal

Local Chair

Ben Juurlink

Delft University of Technology,
The Netherlands**Vice-Chairs**

Alain Darte

CNRS-Lyon, France

Wolfgang Karl

University of Karlsruhe, Germany

Topic 5: Parallel and Distributed Databases**Global Chair**

Alex Szalay

The Johns Hopkins University, USA

Local Chair

Djoerd Hiemstra

University of Twente, The Netherlands

Vice-Chairs

Alfons Kemper

Technische Universität München, Germany

Manuel Prieto

Universidad Complutense Madrid, Spain

Topic 6: Grid, Cluster, and Cloud Computing

Global Chair

Jon Weissman University of Minnesota, USA

Local Chair

Lex Wolters Leiden University, The Netherlands

Vice-Chairs

David Abramson
Marty Humphrey

Topic 7: Peer-to-Peer Computing

Global Chair

Ben Zhao University of California at Santa Barbara, USA

Local Chair

Paweł Garbacki Google Research, Switzerland

Vice-Chairs

Christos Gkantsidis Microsoft Research, UK
Adriana Iamnitchi University of South Florida, USA
Spyros Voulgaris Vrije Universiteit, The Netherlands

Topic 8: Distributed Systems and Algorithms

Global Chair

Dejan Kostić EPFL, Switzerland

Local Chair

Guillaume Pierre Vrije Universiteit, The Netherlands

Vice-Chairs

Flavio Junqueira
Peter R. Pietzuch
Yahoo! Research, Spain
Imperial College London, UK

Topic 9: Parallel and Distributed Programming

Global Chair

Domenico Talia University of Calabria, Italy

Local Chair

Jason Maassen Vrije Universiteit, The Netherlands

Vice-Chairs

Fabrice Huet INRIA-Sophia Antipolis, France
 Shantenu Jha Louisiana State University, USA

Topic 10: Parallel Numerical Algorithms**Global Chair**

Peter Arbenz ETH Zürich, Switzerland

Local Chair

Martin van Gijzen Delft University of Technology,
 The Netherlands

Vice-Chairs

Patrick Amestoy INPT-ENSEEIHT, France
 Pasqua D'Ambrìa ICAR-CNR, Italy

Topic 11: Multicore and Manycore Programming**Global Chair**

Barbara Chapman University of Houston, USA

Local Chair

Bart Kienhuis Leiden University, The Netherlands

Vice-Chairs

Eduard Ayguadé Universitat Politècnica de Catalunya, Spain
 François Bodin Irisa, France
 Oscar Plata University of Malaga, Spain
 Eric Stotzer University of Houston, USA

Topic 12: Theory and Algorithms for Parallel Computation**Global Chair**

Andrea Pietracaprina University of Padova, Italy

Local Chair

Rob Bisseling Utrecht University, The Netherlands

Vice-Chairs

Emmanuelle Lebhar CNRS, France
 Alexander Tiskin University of Warwick, UK

Topic 13: High-Performance Networks

Global Chair

Cees de Laat University of Amsterdam, The Netherlands

Local Chair

Chris Develder Ghent University, Belgium

Vice-Chairs

Admela Jukan
Joe Mambretti
Technische Universität Braunschweig, Germany
Northwestern University, USA

Topic 14: Mobile and Ubiquitous Computing

Global Chair

Gerd Kortuem Lancaster University, UK

Local Chair

Henk Eertink Telematica Instituut, The Netherlands

Vice-Chairs

Christian Prehofer Nokia Research, Finland
Martin Strohbach NEC Europe Ltd., Germany

Euro-Par 2009 Referees

- | | |
|-------------------------|---------------------|
| Imad Aad | Martin Bauer |
| David Abramson | Daniel Becker |
| Cody Addison | Marcel Beemster |
| Marco Aldinucci | Adam Belloum |
| Mauricio Alvarez Mesa | Vicenç Beltran |
| Brian Amedro | Mladen Berekovic |
| Alexander van Amesfoort | Vandy Berten |
| Patrick Amestoy | Robert Birke |
| Dieter an Mey | Sumit Birla |
| Paul Anderson | Eric Biscondi |
| Laura Antonelli | Bartosz Biskupski |
| Gabriel Antoniu | Rob Bisseling |
| Peter Arbenz | Jeremy Blackburn |
| Eduard Ayguadé | François Bodin |
| Rosa Badia | David Boehme |
| Ranieri Baraglia | Matthias Bollhoefer |
| Mortaza Bargh | Raphaël Bolze |
| Umit Batur | Olivier Bonaventure |

Edward Bortnikov
Ivona Brandic
Steven Brandt
René Brunner
Marian Bubak
Mark Bull
Mathijs den Burger
Alfredo Buttari
Surendra Byna
Wolfgang Bziuk
Massimo Cafaro
Berkant Barla Cambazoglu
Kirk Cameron
Louis-Claude Canon
Antonio Cansado
Thomas Carroll
Eduardo Cesar
Pablo Chacin
Mohit Chamanian
Barbara Chapman
Yong Chen
Xiaomin Chen
Shyam Chikatamarla
Dave Clarke
Philippe Clauss
Murray Cole
Stefania Corsaro
Simon Courtenage
Davide Cuda
José Cunha
Salvatore Cuomo
Pasqua D'Ambra
Quan Dang Minh
Francisco de Sande
Bart De Vleeschauwer
Lien Deboosere
Chirag Dekate
Zhigang Deng
Luiz DeRose
Roberto Di Cosmo
Freek Dijkstra
Pedro Diniz
Matt Dobson
Niels Drost
Dominique Dudkowski
Alejandro Duran Gonzalez
Henk Eertink
Erik Elmroth
Dick Epema
Thomas Fahringer
Zhibin Fang
Paul Feautrier
Renato Figueiredo
Dimitrios Filippopoulos
Joshua Finniss
Ciorba Florina Monica
Gianluigi Folino
Daniel Franco
Pierre Francois
Stefan Freitag
Wolfgang Frings
Włodzimierz Funika
Jose Gómez
Edgar Gabriel
Antonia Gallardo Gomez
Sebastia Galmes
Paweł Garbacki
Nandan Garg
Alan Gatherer
Frédéric Gava
Markus Geimer
Krassimir Georgiev
Joseph Gergaud
Michael Gerndt
Abdullah Gharaibeh
Martin van Gijzen
Luca Giraudo
Apostolos Gkamas
Christos Gkantsidis
Marc Gonzalez
Sergei Gorlatch
Serge Gratton
Francesco Gregoretti
Laura Grigori
Christian Grimme
Daniel Grosu
Mario Rosario Guarracino
Abdou Guermouche
Ronan Guiuarch
Eladio Gutierrez

Martin Gutknecht
Soonhoi Ha
Phuong Ha
Azzam Haidar
Jeroen van der Ham
Djoerd Hiemstra
Daniel Higuero
Zach Hill
Zhenjiang Hu
Lei Huang
Kevin Huck
Fabrice Huet
Bob Hulsebosch
Marty Humphrey
Adriana Iamnitchi
Yves Ineichen
Alexandru Iosup
Ceriel Jacobs
Heike Jagode
William Jalby
Emmanuel Jeannot
Shantenu Jha
Bin Jiang
Josep Jorba
Mackale Joyner
Admela Jukan
Flavio Junqueira
Ben Juurlink
Indira Kapoor
Helen Karatza
Wolfgang Karl
Sven Karlsson
Rainer Keller
Alfons Kemper
Maurice van Keulen
Thilo Kielmann
Bart Kienhuis
Zach King
Nikola Knezevic
Vladimir Korkhov
Gerd Kortuem
Dejan Kostić
Nicolas Kourtellis
Gayathri Krishnamurthy
Jean-Yves L'Excellent
Per Lótstedt
Piero Lanucara
Bertrand Le Gal
Xavier León Gutiérrez
Cees de Laat
Emmanuelle Lebhar
Jonathan Ledlie
Vincent Lefèvre
Virginie Legrand
Francesco Leporati
Joachim Lepping
Mario Leyton
Jie Li
Hai-Xiang Lin
Sander van der Maar
Jason Maassen
Joel Mambretti
Pierre Manneback
Ming Mao
Tomàs Margalef
Zelda Marino
Xavier Martorell
Carlo Mastroianni
Constandinos Mavromoustakis
Eduard Mehofer
Sjoerd Meijer
Celso Mendes
Pascal Mérindol
Andre Merzky
Peter Messmer
Pierre Michaud
Geyong Min
Peter Minev
Raffaele Montella
Anna Morajko
Juan Carlos Moure
Sandrine Mouysset
Fernando Mujica
Dmitry Nadezhkin
Vijay Naik
Thoai Nam
Jeff Napper
Rob van Nieuwpoort
Dimitrios Nikolopoulos
Rajesh Nishtala

Daniel Nurmi
Gabriel Oksa
Gennaro Oliva
Alexander Papaspyrou
Marcin Paprzycki
Sang-Min Park
Satish Penmatsa
Wesley Petersen
Michele Petracca
Alan Phipps
Luciano Piccoli
Guillaume Pierre
Andrea Pietracaprina
Peter Pietzuch
Andy Pimentel
Oscar Plata
Sabri Pllana
Ronald van der Pol
Nathanaël Prémillieu
Christian Prehofer
Lydia Prieto
Manuel Prieto Matias
Krishna Puttaswamy
Bart Puype
Martin Quinson
Thomas Röblitz
Sanjay Rajopadhye
Govindarajan Ramaswamy
Enrico Rantala
Benjamin Reed
Kees van Reeuwijk
Matei Ripeanu
Alma Riska
Marcela Rivera
Ivan Rodero
Jeff Rose
Horacio Rostro-Gonzalez
Philip Roth
Sergio Rovida
Hana Rudová
Arkaitz Ruiz-Alvarez
Jussi Ruutu
Marc Sánchez Artigas
Renata Slota
Gianni Sacchi
Jan Sacha
Ponnuswamy Sadayappan
Alessandra Sala
Frode Eika Sandnes
Jagadeesh Sankaran
Guna Santos
Elizeu Santos-Neto
Erik Saule
Olaf Schenk
Florian Schintke
Erik Schnetter
Simon Schubert
Martin Schulz
Frank Seinstra
Ashish Shrivastava
Federico Silla
Henk Sips
David Skillicorn
Todd Snider
Ozan Sonmez
Daniel Soto
Florent Sourbier
Giandomenico Spezzano
RosaMaria Spitaleri
Tim Stevens
Eric Stotzer
Corina Stratan
Martin Strohbach
Dineel Sule
Xian-He Sun
Alan Sussman
Frederic Suter
Alex Szalay
Zoltan Szebenyi
Michal Szymaniak
Danesh Tafti
Toktam Taghavi
Domenico Talia
Andrei Tchernykh
Christian Tenllado
Eno Thereska
Juan Manuel Tirado
Alexander Tiskin
Nicola Tonellotto
Johan Tordsson

XVIII Organization

Paolo Trunfio
Hong-Linh Truong
Bora Ucar
Benny Van Houdt
Ana Lucia Varbanescu
George Varsamopoulos
Nedeljko Vasic
Xavier Vasseur
Kees Verstoep
Christof Voemel
Spyros Voulgaris
Arjen de Vries
John Paul Walters
Jon Weissman
Dirk Westhoff
Philipp Wieder
Christo Wilson
Marcus Wittberger
Felix Wolf
Lex Wolters
Xingfu Wu
Brian Wylie
Maysam Yabandeh
Kun Yang
Albert-Jan Yzelman
Sharrukh Zaman
Frank Zdarsky
Li Zhang
Ben Zhao
Wolfgang Ziegler
Eugenio Zimeo

Table of Contents

Abstracts Invited Talks

Multicore Programming Challenges	1
<i>Michael Perrone</i>	
Ibis: A Programming System for Real-World Distributed Computing	3
<i>Henri Bal</i>	
What Is in a Namespace?	4
<i>Antony Rowstron</i>	

Topic 1: Support Tools and Environments

Introduction	7
<i>Felix Wolf, Andy D. Pimentel, Luiz DeRose, Soonhoi Ha, Thilo Kielmann, and Anna Morajko (Topic Chairs)</i>	
Atune-IL: An Instrumentation Language for Auto-tuning Parallel Applications	9
<i>Christoph A. Schaefer, Victor Pankratius, and Walter F. Tichy</i>	
Assigning Blame: Mapping Performance to High Level Parallel Programming Abstractions	21
<i>Nick Rutar and Jeffrey K. Hollingsworth</i>	
A Holistic Approach towards Automated Performance Analysis and Tuning	33
<i>Guogjing Cong, I-Hsin Chung, Huifang Wen, David Klepacki, Hiroki Murata, Yasushi Negishi, and Takao Moriyama</i>	
Pattern Matching and I/O Replay for POSIX I/O in Parallel Programs	45
<i>Michael Kluge, Andreas Knüpfer, Matthias Müller, and Wolfgang E. Nagel</i>	
An Extensible I/O Performance Analysis Framework for Distributed Environments	57
<i>Benjamin Eckart, Xubin He, Hong Ong, and Stephen L. Scott</i>	
Grouping MPI Processes for Partial Checkpoint and Co-migration	69
<i>Rajendra Singh and Peter Graham</i>	
Process Mapping for MPI Collective Communications	81
<i>Jin Zhang, Jidong Zhai, Wenguang Chen, and Weimin Zheng</i>	

Topic 2: Performance Prediction and Evaluation

Introduction	95
<i>Thomas Fahringer, Alexandru Iosup, Marian Bubak, Matei Ripeanu, Xian-He Sun, and Hong-Linh Truong (Topic Chairs)</i>	
Stochastic Analysis of Hierarchical Publish/Subscribe Systems	97
<i>Gero Mühl, Arnd Schröter, Helge Parzy jegla, Samuel Kounev, and Jan Richling</i>	
Characterizing and Understanding the Bandwidth Behavior of Workloads on Multi-core Processors	110
<i>Guoping Long, Dongrui Fan, and Junchao Zhang</i>	
Hybrid Techniques for Fast Multicore Simulation	122
<i>Manu Shantharam, Padma Raghavan, and Mahmut Kandemir</i>	
PSINS: An Open Source Event Tracer and Execution Simulator for MPI Applications	135
<i>Mustafa M. Tikir, Michael A. Laurenzano, Laura Carrington, and Allan Snavely</i>	
A Methodology to Characterize Critical Section Bottlenecks in DSM Multiprocessors	149
<i>Benjamín Sahelices, Pablo Ibáñez, Víctor Viñals, and J.M. Llabería</i>	

Topic 3: Scheduling and Load Balancing

Introduction	165
<i>Emmanuel Jeannot, Ramin Yahyapour, Daniel Grosu, and Helen Karatza (Topic Chairs)</i>	
Dynamic Load Balancing of Matrix-Vector Multiplications on Roadrunner Compute Nodes	166
<i>José Carlos Sancho and Darren J. Kerbyson</i>	
A Unified Framework for Load Distribution and Fault-Tolerance of Application Servers	178
<i>Huaigu Wu and Bettina Kemme</i>	
On the Feasibility of Dynamically Scheduling DAG Applications on Shared Heterogeneous Systems	191
<i>Aline P. Nascimento, Alexandre Sena, Cristina Boeres, and Vinod E.F. Rebbello</i>	
Steady-State for Batches of Identical Task Trees	203
<i>Sékou Diakité, Loris Marchal, Jean-Marc Nicod, and Laurent Philippe</i>	

A Buffer Space Optimal Solution for Re-establishing the Packet Order in a MPSoC Network Processor	216
<i>Daniela Genius, Alix Munier Kordon, and Khouloud Zine el Abidine</i>	
Using Multicast Transfers in the Replica Migration Problem: Formulation and Scheduling Heuristics	228
<i>Nikos Tziritas, Thanasis Loukopoulos, Petros Lampsas, and Spyros Lalis</i>	
A New Genetic Algorithm for Scheduling for Large Communication Delays	241
<i>Johnatan E. Pecero, Denis Trystram, and Albert Y. Zomaya</i>	
Comparison of Access Policies for Replica Placement in Tree Networks	253
<i>Anne Benoit</i>	
Scheduling Recurrent Precedence-Constrained Task Graphs on a Symmetric Shared-Memory Multiprocessor	265
<i>UmaMaheswari C. Devi</i>	
Energy-Aware Scheduling of Flow Applications on Master-Worker Platforms	281
<i>Jean-François Pineau, Yves Robert, and Frédéric Vivien</i>	
Topic 4: High Performance Architectures and Compilers	
Introduction	295
<i>Pedro C. Diniz, Ben Juurlink, Alain Darte, and Wolfgang Karl (Topic Chairs)</i>	
Last Bank: Dealing with Address Reuse in Non-Uniform Cache Architecture for CMPs	297
<i>Javier Lira, Carlos Molina, and Antonio González</i>	
Paired ROBs: A Cost-Effective Reorder Buffer Sharing Strategy for SMT Processors	309
<i>R. Ubal, J. Sahuquillo, S. Petit, and P. López</i>	
REPAS: Reliable Execution for Parallel ApplicationS in Tiled-CMPs ...	321
<i>Daniel Sánchez, Juan L. Aragón, and José M. García</i>	
Impact of Quad-Core Cray XT4 System and Software Stack on Scientific Computation	334
<i>S.R. Alam, R.F. Barrett, H. Jagode, J.A. Kuehn, S.W. Poole, and R. Sankaran</i>	

Topic 5: Parallel and Distributed Databases

Introduction	347
<i>Alex Szalay, Djoerd Hiemstra, Alfons Kemper, and Manuel Prieto (Topic Chairs)</i>	
Unifying Memory and Database Transactions	349
<i>Ricardo J. Dias and João M. Lourenço</i>	
A DHT Key-Value Storage System with Carrier Grade Performance	361
<i>Guangyu Shi, Jian Chen, Hao Gong, Lingyuan Fan, Haiqiang Xue, Qingming Lu, and Liang Liang</i>	
Selective Replicated Declustering for Arbitrary Queries	375
<i>K. Yasin Oktay, Ata Turk, and Cevdet Aykanat</i>	

Topic 6: Grid, Cluster, and Cloud Computing

Introduction	389
<i>Jon Weissman, Lex Wolters, David Abramson, and Marty Humphrey (Topic Chairs)</i>	
POGGI: Puzzle-Based Online Games on Grid Infrastructures	390
<i>Alexandru Iosup</i>	
Enabling High Data Throughput in Desktop Grids through Decentralized Data and Metadata Management: The BlobSeer Approach	404
<i>Bogdan Nicolae, Gabriel Antoniu, and Luc Bougé</i>	
MapReduce Programming Model for .NET-Based Cloud Computing	417
<i>Chao Jin and Rajkumar Buyya</i>	
The Architecture of the XtreemOS Grid Checkpointing Service	429
<i>John Mehnert-Spahn, Thomas Ropars, Michael Schoettner, and Christine Morin</i>	
Scalable Transactions for Web Applications in the Cloud	442
<i>Zhou Wei, Guillaume Pierre, and Chi-Hung Chi</i>	
Provider-Independent Use of the Cloud	454
<i>Terence Harmer, Peter Wright, Christina Cunningham, and Ron Perrott</i>	
MPI Applications on Grids: A Topology Aware Approach	466
<i>Camille Coti, Thomas Herault, and Franck Cappello</i>	

Topic 7: Peer-to-Peer Computing

Introduction	481
<i>Ben Zhao, Paweł Garbacki, Christos Gkantsidis, Adriana Iamnitchi, and Spyros Voulgaris (Topic Chairs)</i>	

A Least-Resistance Path in Reasoning about Unstructured Overlay Networks	483
<i>Giorgos Georgiadis and Marina Papatriantafilou</i>	
SiMPSON: Efficient Similarity Search in Metric Spaces over P2P Structured Overlay Networks	498
<i>Quang Hieu Vu, Mihai Lupu, and Sai Wu</i>	
Uniform Sampling for Directed P2P Networks	511
<i>Cyrus Hall and Antonio Carzaniga</i>	
Adaptive Peer Sampling with Newscast	523
<i>Norbert Tölgysesi and Márk Jelasity</i>	
Exploring the Feasibility of Reputation Models for Improving P2P Routing under Churn	535
<i>Marc Sánchez-Artigas, Pedro García-López, and Blas Herrera</i>	
Selfish Neighbor Selection in Peer-to-Peer Backup and Storage Applications	548
<i>Pietro Michiardi and Laszlo Toka</i>	
Zero-Day Reconciliation of BitTorrent Users with Their ISPs	561
<i>Marco Slot, Paolo Costa, Guillaume Pierre, and Vivek Rai</i>	
Surfing Peer-to-Peer IPTV: Distributed Channel Switching	574
<i>A.-M. Kermarrec, E. Le Merrer, Y. Liu, and G. Simon</i>	
Topic 8: Distributed Systems and Algorithms	
Introduction	589
<i>Dejan Kostić, Guillaume Pierre, Flávio Junqueira, and Peter R. Pietzuch (Topic Chairs)</i>	
Distributed Individual-Based Simulation	590
<i>Jiming Liu, Michael B. Dillencourt, Lubomir F. Bic, Daniel Gillen, and Arthur D. Lander</i>	
A Self-stabilizing K-Clustering Algorithm Using an Arbitrary Metric	602
<i>Eddy Caron, Ajoy K. Datta, Benjamin Depardon, and Lawrence L. Larmore</i>	
Active Optimistic Message Logging for Reliable Execution of MPI Applications	615
<i>Thomas Ropars and Christine Morin</i>	
Topic 9: Parallel and Distributed Programming	
Introduction	629
<i>Domenico Talia, Jason Maassen, Fabrice Huet, and Shantenu Jha (Topic Chairs)</i>	

A Parallel Numerical Library for UPC	630
<i>Jorge González-Domínguez, María J. Martín, Guillermo L. Taboada, Juan Touriño, Ramón Doallo, and Andrés Gómez</i>	
A Multilevel Parallelization Framework for High-Order Stencil Computations	642
<i>Hikmet Dursun, Ken-ichi Nomura, Liu Peng, Richard Seymour, Weiqiang Wang, Rajiv K. Kalia, Aiichiro Nakano, and Priya Vashishta</i>	
Using OpenMP vs. Threading Building Blocks for Medical Imaging on Multi-cores	654
<i>Philipp Kegel, Maraike Schellmann, and Sergei Gorlatch</i>	
Parallel Skeletons for Variable-Length Lists in SkeTo Skeleton Library	666
<i>Haruto Tanno and Hideya Iwasaki</i>	
STKM on SCA: A Unified Framework with Components, Workflows and Algorithmic Skeletons	678
<i>Marco Aldinucci, Hinde Lilia Bouziane, Marco Danelutto, and Christian Pérez</i>	
Grid-Enabling SPMD Applications through Hierarchical Partitioning and a Component-Based Runtime	691
<i>Elton Mathias, Vincent Cavé, Stéphane Lanteri, and Françoise Baude</i>	
Reducing Rollbacks of Transactional Memory Using Ordered Shared Locks	704
<i>Ken Mizuno, Takuya Nakaike, and Toshio Nakatani</i>	

Topic 10: Parallel Numerical Algorithms

Introduction	719
<i>Peter Arbenz, Martin van Gijzen, Patrick Amestoy, and Pasqua D'Ambra (Topic Chairs)</i>	
Wavelet-Based Adaptive Solvers on Multi-core Architectures for the Simulation of Complex Systems	721
<i>Diego Rossinelli, Michael Bergdorf, Babak Hejazialhosseini, and Petros Koumoutsakos</i>	
Localized Parallel Algorithm for Bubble Coalescence in Free Surface Lattice-Boltzmann Method	735
<i>Stefan Donath, Christian Feichtinger, Thomas Pohl, Jan Götz, and Ulrich Rüde</i>	

Fast Implicit Simulation of Oscillatory Flow in Human Abdominal Bifurcation Using a Schur Complement Preconditioner	747
<i>K. Burckhardt, D. Szczerba, J. Brown, K. Muralidhar, and G. Székely</i>	
A Parallel Rigid Body Dynamics Algorithm	760
<i>Klaus Iglberger and Ulrich Rüde</i>	
Optimized Stencil Computation Using In-Place Calculation on Modern Multicore Systems	772
<i>Werner Augustin, Vincent Heuveline, and Jan-Philipp Weiss</i>	
Parallel Implementation of Runge–Kutta Integrators with Low Storage Requirements	785
<i>Matthias Korch and Thomas Rauber</i>	
PSPIKE: A Parallel Hybrid Sparse Linear System Solver	797
<i>Murat Manguoglu, Ahmed H. Sameh, and Olaf Schenk</i>	
Out-of-Core Computation of the QR Factorization on Multi-core Processors	809
<i>Mercedes Marqués, Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, and Robert van de Geijn</i>	
Adaptive Parallel Householder Bidiagonalization	821
<i>Fangbin Liu and Frank J. Seinstra</i>	

Topic 11: Multicore and Manycore Programming

Introduction	837
<i>Barbara Chapman, Bart Kienhuis, Eduard Ayguadé, François Bodin, Oscar Plata, and Eric Stotzer (Topic Chairs)</i>	
Tile Percolation: An OpenMP Tile Aware Parallelization Technique for the Cyclops-64 Multicore Processor	839
<i>Ge Gan, Xu Wang, Joseph Manzano, and Guang R. Gao</i>	
An Extension of the StarSs Programming Model for Platforms with Multiple GPUs	851
<i>Eduard Ayguadé, Rosa M. Badia, Francisco D. Igual, Jesús Labarta, Rafael Mayo, and Enrique S. Quintana-Ortí</i>	
STARPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures	863
<i>Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier</i>	
XJava: Exploiting Parallelism with Object-Oriented Stream Programming	875
<i>Frank Otto, Victor Pankratius, and Walter F. Tichy</i>	

JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA	887
<i>Yonghong Yan, Max Grossman, and Vivek Sarkar</i>	
Fast and Efficient Synchronization and Communication Collective Primitives for Dual Cell-Based Blades	900
<i>Epifanio Gaona, Juan Fernández, and Manuel E. Acacio</i>	
Searching for Concurrent Design Patterns in Video Games	912
<i>Micah J. Best, Alexandra Fedorova, Ryan Dickie, Andrea Tagliasacchi, Alex Couture-Beil, Craig Mustard, Shane Mottishaw, Aron Brown, Zhi Feng Huang, Xiaoyuan Xu, Nasser Ghazali, and Andrew Brownsword</i>	
Parallelization of a Video Segmentation Algorithm on CUDA-Enabled Graphics Processing Units	924
<i>Juan Gómez-Luna, José María González-Linares, José Ignacio Benavides, and Nicolás Guil</i>	
A Parallel Point Matching Algorithm for Landmark Based Image Registration Using Multicore Platform	936
<i>Lin Yang, Leiguang Gong, Hong Zhang, John L. Noshier, and David J. Foran</i>	
High Performance Matrix Multiplication on Many Cores	948
<i>Nan Yuan, Yongbin Zhou, Guangming Tan, Junchao Zhang, and Dongrui Fan</i>	
Parallel Lattice Basis Reduction Using a Multi-threaded Schnorr-Euchner LLL Algorithm	960
<i>Werner Backes and Susanne Wetzel</i>	
Efficient Parallel Implementation of Evolutionary Algorithms on GPGPU Cards	974
<i>Ogier Maitre, Nicolas Lachiche, Philippe Clauss, Laurent Baumes, Avelino Corma, and Pierre Collet</i>	
Topic 12: Theory and Algorithms for Parallel Computation	
Introduction	989
<i>Andrea Pietracaprina, Rob Bisseling, Emmanuelle Lebhar, and Alexander Tiskin (Topic Chairs)</i>	
Implementing Parallel Google Map-Reduce in Eden	990
<i>Jost Berthold, Mischa Dieterle, and Rita Loogen</i>	
A Lower Bound for Oblivious Dimensional Routing	1003
<i>Andre Osterloh</i>	

Topic 13: High-Performance Networks

Introduction	1013
<i>Cees de Laat, Chris Develder, Admela Jukan, and Joe Mambretti (Topic Chairs)</i>	
A Case Study of Communication Optimizations on 3D Mesh Interconnects	1015
<i>Abhinav Bhatelé, Eric Bohm, and Laxmikant V. Kalé</i>	
Implementing a Change Assimilation Mechanism for Source Routing Interconnects	1029
<i>Antonio Robles-Gómez, Aurelio Bermúdez, and Rafael Casado</i>	
Dependability Analysis of a Fault-Tolerant Network Reconfiguring Strategy	1040
<i>Vicente Chirivella, Rosa Alcover, José Flich, and José Duato</i>	
RecTOR: A New and Efficient Method for Dynamic Network Reconfiguration	1052
<i>Åshild Grønstad Solheim, Olav Lysne, and Tor Skeie</i>	
NIC-Assisted Cache-Efficient Receive Stack for Message Passing over Ethernet	1065
<i>Brice Goglin</i>	
A Multipath Fault-Tolerant Routing Method for High-Speed Interconnection Networks	1078
<i>Gonzalo Zarza, Diego Lugones, Daniel Franco, and Emilio Luque</i>	
Hardware Implementation Study of the SCFQ-CA and DRR-CA Scheduling Algorithms	1089
<i>Raúl Martínez, Francisco J. Alfaro, José L. Sánchez, and José M. Claver</i>	

Topic 14: Mobile and Ubiquitous Computing

Introduction	1103
<i>Gerd Kortuem, Henk Eertink, Christian Prehofer, and Martin Strohbach (Topic Chairs)</i>	
Optimal and Near-Optimal Energy-Efficient Broadcasting in Wireless Networks	1104
<i>Christos A. Papageorgiou, Panagiotis C. Kokkinos, and Emmanouel A. Varvarigos</i>	

Author Index	1117
--------------------	------

Multicore Programming Challenges

Michael Perrone^{*}

IBM, TJ Watson Research Lab
Yorktown Heights, NY, USA
mpp@us.ibm.com

Abstract. The computer industry is facing fundamental challenges that are driving a major change in the design of computer processors. Due to restrictions imposed by quantum physics, one historical path to higher computer processor performance - by increased clock frequency - has come to an end. Increasing clock frequency now leads to power consumption costs that are too high to justify. As a result, we have seen in recent years that the processor frequencies have peaked and are receding from their high point. At the same time, competitive market conditions are giving business advantage to those companies that can field new streaming applications, handle larger data sets, and update their models to market conditions faster. The desire for newer, faster and larger is driving continued demand for higher computer performance.

The industry's response to address these challenges has been to embrace "multicore" technology by designing processors that have multiple processing cores on each silicon chip. Increasing the number of cores per chip has enable processor peak performance to double with each doubling of the number of cores. With performance doubling occurring at approximately constant clock frequency so that energy costs can be controlled, multicore technology is poised to deliver the performance users need for their next generation applications while at the same time reducing total cost of ownership per FLOP.

The multicore solution to the clock frequency problem comes at a cost: Performance scaling on multicore is generally sub-linear and frequently decreases beyond some number of cores. For a variety of technical reasons, off-chip bandwidth is not increasing as fast as the number of cores per chip which is making memory and communication bottlenecks the main barrier to improved performance. What these bottlenecks mean to multicore users is that precise and flexible control of data flows will be crucial to achieving high performance. Simple mappings of their existing algorithms to multicore will not result in the naive performance scaling one might expect from increasing the number of cores per chip. Algorithmic changes, in many cases major, will have to be made to get value out of multicore. Multicore users will have to re-think and in many cases re-write their applications if they want to achieve high performance. Multicore forces each programmer to become a parallel programmer; to think of their chips as clusters; and to deal with the issues of communication, synchronization, data transfer and nondeterminism as integral elements

^{*} Invited Speaker.

of their algorithms. And for those already familiar with parallel programming, multicore processors add a new level of parallelism and additional layers of complexity.

This talk will highlight some of the challenges that need to be overcome in order to get better performance scaling on multicore, and will suggest some solutions.

Ibis: A Programming System for Real-World Distributed Computing

Henri Bal*

Vrije Universiteit
Amsterdam, The Netherlands
`bal@cs.vu.nl`

Abstract. The landscape of distributed computing systems has changed many times over the previous decades. Modern real-world distributed systems consist of clusters, grids, clouds, desktop grids, and mobile devices. Writing applications for such systems has become increasingly difficult. The aim of the Ibis project is to drastically simplify the programming of such applications. The Ibis philosophy is that real-world distributed applications should be developed and compiled on a local workstation, and simply be launched from there.

The Ibis project studies several fundamental problems of distributed computing hand-in-hand with major applications, and integrates the various solutions in one programming system. Applications that drive the Ibis project include scientific ones (in the context of the VL-e project), multimedia content analysis, distributed reasoning, and many others. The fundamental problems we study include performance, heterogeneity, malleability, fault-tolerance, and connectivity. Solutions to these fundamental problems are integrated in the Ibis programming system, which is written entirely in Java. Examples are: runtime systems for many programming models (divide-and-conquer, master-worker, etc.), a Java-centric communication library (IPL), a library that automatically solves connectivity problems (SmartSockets), a graphical deployment system, a library that hides the underlying middlewares (JavaGAT), and a simple peer-to-peer middleware (Zorilla). The resulting system has been used in several award-winning competitions, including SCALE 2008 (at CCgrid'08) and DACH 2008 (at Cluster/Grid'08).

The talk describes the rationale behind Ibis, the main applications (in particular multimedia content analysis), and its most important software components. Next, it discusses experiments where Ibis is used to run parallel applications efficiently on a large-scale distributed system consisting of several clusters, a desktop grid, and an Amazon cloud. Finally, the talk discusses recent work in using Ibis for distributed applications that run on mobile devices (smart phones) or that deploy distributed applications from a mobile device. An example is given where a distributed object-recognition application is launched from an Android phone, so the user can perform tasks that would be impossible on the limited resources of the phone itself.

For more information on Ibis, please see <http://www.cs.vu.nl/ibis/>.

* Invited Speaker.

What Is in a Namespace?

Antony Rowstron*

Microsoft Research
Cambridge, UK
antr@microsoft.com

Abstract. Building large-scale distributed systems that are decentralized is challenging. Distributed Hash Tables, or structured overlays, attempted to make the process of building these distributed systems easier, by abstracting away many of the complexities of the physical topology, and provide a virtual namespace. The virtual namespace can then be exploited, for example to build or maintain other data structures like a tree to support application-level multicast or group communication. Because the namespace is independent of the underlying physical topology, it can be used to help mitigate the impact of network topology change, for example through churn. A challenge is to understand how to exploit the difference between a physical topology and a virtual topology (or namespace). In the first half of the talk, using examples from a number of systems I have worked on over the years, I will highlight how a namespace can be used and the differences between algorithms designed to work in virtual versus physical topologies.

In the second half of the talk I will then outline how recently we have begun more deeply exploring the relationship between physical and virtual topologies. The first is exploiting a virtual topology to enable disruption-tolerant routing protocols that can work on networks that are frequently and dynamically partitioning. The second is examining how we can build physical topologies that are more aligned to a virtual topology, and how this makes building servers that run in Internet-scale Data Centres easier to implement.

* Invited Speaker.

Topic 1

Support Tools and Environments

Introduction

Felix Wolf*, Andy D. Pimentel*, Luiz DeRose*, Soonhoi Ha*,
Thilo Kielmann*, and Anna Morajko*

The spread of systems that provide parallelism either “in-the-large” (grid infrastructures, clusters) or “in-the-small” (multi-core chips) creates new opportunities for exploiting parallelism in a wider spectrum of application domains. However, the increasing complexity of parallel and distributed platforms renders the programming, the use, and the management of these systems a costly endeavor that requires advanced expertise and skills. There is therefore an increasing need for powerful support tools and environments that will help end users, application programmers, software engineers and system administrators to manage the increasing complexity of parallel and distributed platforms. This topic aims at bringing together tool designers, developers, and users in order to share novel ideas, concepts, and products covering a wide range of platforms, including homogeneous and heterogeneous multicore architectures. The Program Committee sought high-quality contributions with solid foundations and experimental validations on real systems, and encouraged the submission of new ideas on intelligent monitoring and diagnosis tools and environments which can exploit behavior knowledge to detect programming bugs or performance bottlenecks and help ensure correct and efficient parallel program execution.

This year, twenty-one papers were submitted to Topic 1 of which seven papers were accepted. The accepted papers cover a wide range of interests and contributions:

- Two papers study the performance analysis, profiling, and benchmarking of I/O. The paper “Pattern Matching and I/O Replay for POSIX I/O in Parallel Programs” describes an approach for application-oriented file system benchmarking, which allows the I/O behavior of parallel applications to be tracked and replayed. The paper “An Extensible I/O Performance Analysis Framework for Distributed Environments” presents ExPerT, an Extensible Performance Toolkit. ExPerT defines a flexible framework from which a set of benchmarking, tracing, and profiling applications can be correlated together in a unified interface.
- Two papers address the performance analysis and tuning of parallel applications. The paper “Atune-IL: An Instrumentation Language for Auto-Tuning Parallel Applications” introduces Atune-IL, an instrumentation language that uses new types of code annotations to mark tuning parameters, blocks, permutation regions, or measuring points. The paper “A Holistic Approach towards Automated Performance Analysis and Tuning” aims at improving the productivity of performance debugging of parallel applications. To this end,

* Topic Chairs.

it proposes a framework that facilitates the combination of expert knowledge, compiler techniques, and performance research for performance diagnosis and solution discovery.

- The paper “Assigning Blame: Mapping Performance to High Level Parallel Programming Abstractions” discusses mapping mechanisms, called variable blame, for creating mappings of high-level programming abstractions down to the low-level parallel programming constructs and using them to assist in the profiling and debugging of programs created using advanced parallel programming techniques.
- The paper “Process Mapping for Collective Communications” proposes a way to handle collective communications which transforms them into a series of point-to-point communication operations according to their implementation in communication libraries. Hereafter, existing approaches can be used to find optimized mapping schemes which are optimized for both point-to-point and collective communications.
- The paper “Grouping MPI Processes for Partial Checkpoint and co-Migration” studies a partial checkpoint and migrate facility which allows for checkpointing and migrating only a subset of the running MPI processes while the others can continue to execute and make progress.

Atune-IL: An Instrumentation Language for Auto-tuning Parallel Applications

Christoph A. Schaefer, Victor Pankratius, and Walter F. Tichy

University of Karlsruhe (TH), Am Fasanengarten 5, 76131 Karlsruhe, Germany
`{cschaefer,pankratius,tichy}@ipd.uka.de`

Abstract. Auto-tuners automate the performance tuning of parallel applications. Three major drawbacks of current approaches are 1) they mainly focus on numerical software; 2) they typically do not attempt to reduce the large search space before search algorithms are applied; 3) the means to provide an auto-tuner with additional information to improve tuning are limited.

Our paper tackles these problems in a novel way by focusing on the interaction between an auto-tuner and a parallel application. In particular, we introduce Atune-IL, an instrumentation language that uses new types of code annotations to mark tuning parameters, blocks, permutation regions, and measuring points. Atune-IL allows a more accurate extraction of meta-information to help an auto-tuner prune the search space *before* employing search algorithms. In addition, Atune-IL's concepts target parallel applications in general, not just numerical programs.

Atune-IL has been successfully evaluated in several case studies with parallel applications differing in size, programming language, and application domain; one case study employed a large commercial application with nested parallelism. On average, Atune-IL reduced search spaces by 78%. In two corner cases, 99% of the search space could be pruned.

1 Introduction

As multicore platforms become ubiquitous, many software applications have to be parallelized and tuned for performance. Manual tuning must be automated to cope with the diversity of application areas for parallelism and the variety of available platforms [1].

Search-based automatic performance tuning (auto-tuning) [2,3,4] is a promising systematic approach for parallel applications. In a repetitive cycle, an auto-tuner executes a parameterized application, monitors it, and modifies parameter values to find a configuration that yields the best performance. It is currently a challenge to specify meta-information for auto-tuning in an efficient and portable way. As search spaces can be large, this additional information can be used an auto-tuner for pruning before applying any search algorithms.

In this paper, we introduce Atune-IL, a general-purpose instrumentation language for auto-tuning. We describe the features and highlight Atune-IL's approach to enrich a program with annotations that allow a more accurate search

space reduction. In several case studies we show how Atune-IL works with different parallel applications and evaluate its effectiveness regarding search space reduction. Finally, we compare Atune-IL in the context of related work.

2 Requirements for a Tuning Instrumentation Language

A tuning instrumentation language for parallel applications should be able to instrument performance relevant variables such that their values can be set by an auto-tuner. It should also allow the demarcation of permutable program statements and provide the means to define program variants; for tuning, these variants represent alternatives of the same program that differ in performance-relevant aspects. Monitoring support is required to give run-time feedback to an auto-tuner, so that it can calculate and try out a new parameter configuration. It is desirable that the tuning language is as general as possible and not closely tied to a particular application domain. We further show that is also important to design the instrumentation language in a way that helps cut down the search space of configurations at an early stage.

2.1 Reducing Search Space before Tuning

The search space an auto-tuner explores is defined by the cross-product of the domains of all parameters within the program. As the search space grows exponentially, even an auto-tuner using a smart search algorithm may need a long time to find the best – or at least a sufficiently good – parameter configuration. For example, consider the search space consisting of 24 million parameter configurations of our first case study in Section 5. If a sophisticated search algorithm tests only 1% of the configurations, it will still perform 240,000 tuning iterations.

We therefore propose reducing the search space *before* any search algorithm is applied. To do so, our instrumentation language is able to capture the structure of an application along with characteristics important for tuning, such as parallel sections and dependencies between tuning parameters.

Analyzing Parallel Sections. Within application code, we define two sections to be independent of each other if they cannot be executed concurrently in any of the application’s execution paths. Nested sections always depend on the enclosing section in that their execution and performance can be influenced by parameters of the enclosing section.

For illustration, consider a hypothetical program as in Fig. 1 with two parallel sections. The sections are independent and cannot run concurrently. Section P_1 has three tuning parameters, t_1, \dots, t_3 , while section P_2 contains two tuning parameters, t_4 and t_5 . An auto-tuner would have to search in the worst case the cross product of all parameters $dom(t_1) \times \dots \times dom(t_5)$. However, if the two sections are known to be independent, the search space can be reduced to the considerably smaller cross products of each section’s parameters $dom(t_1) \times \dots \times dom(t_3)$, and $dom(t_4) \times dom(t_5)$, respectively. The sections can be modularly tuned one

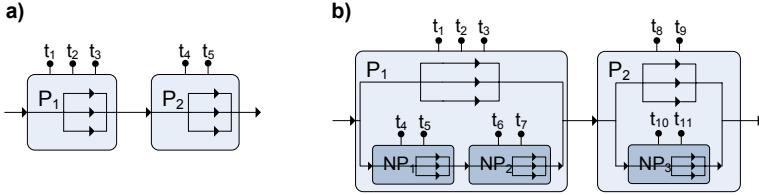


Fig. 1. a) Example of two independent sections with individual tuning parameters; b) example of two nested sections that come with additional parameters

after another in separate tuning sessions. The parameters of the section that is currently not tuned are set to their default values.

Fig. 1 b) extends the previous example and shows P_1 and P_2 , now with the nested sections NP_1 , NP_2 , and NP_3 . To reduce search space in the case of nested structures, the following strategy seemed promising in our case studies. We perform several tuning sessions in which each lowest-level section is tuned together with all parent sections. In this example, this results in $dom(t_1) \times \dots \times dom(t_5)$, $dom(t_1) \times dom(t_2) \times dom(t_3) \times dom(t_6) \times dom(t_7)$, and $dom(t_8) \times \dots \times dom(t_{11})$. If a section is tuned in more than one tuning session (e.g., P_1), its optimal parameter values may differ depending on the nested section chosen before (NP_1 or NP_2). In this case, we tune the parameters of P_1 again and set the parameters of NP_1 and NP_2 to their best values so far. However, this is just an illustration that the information provided by our instrumentation language is useful for an auto-tuner; detailed algorithms are beyond the scope of this paper.

Considering Parameter Dependencies. A tuning parameter often depends on values of another parameter. That is, a particular parameter has to be tuned only if another parameter has some specific value. As an example, consider two parameters named *sortAlgo* and *depth*. The parameter *sortAlgo* selects either parallel merge sort or heap sort, while *depth* defines the depth of recursion within merge sort. Obviously, *depth* conditionally depends on *sortAlgo*. This information is important for an auto-tuner to avoid tuning unnecessary value combinations, as *depth* needs not be tuned for heap sort.

In summary, our instrumentation language makes the search space smaller so that even complex parallel programs can be auto-tuned.

3 Language Features of Atune-IL

This section introduces Atune-IL’s features. We describe how to specify tuning parameters, statement permutation, blocks, and measuring points to meet the requirements introduced in the previous section. For a complete list of all language features refer to Table 1. Listing 1 shows a program instrumented with Atune-IL. The program basically searches strings in a text, stores them in an array and sorts it using parallel sorting algorithms. Finally, it counts the total characters the array contains.

Listing 1. C# program instrumented with Atune-IL

```

List<string> words = new List<string>(3);
void main() {
    #pragma atune startblock fillBlock
    #pragma atune gauge mySortExecTime

    string text = "Auto-tuning has nothing to do with tuning cars."

    #pragma atune startpermutation fillOrder
    #pragma atune nextelem
    words.Add(text.Find("cars"));
    #pragma atune nextelem
    words.Add(text.Find("do"));
    #pragma atune nextelem
    words.Add(text.Find("Auto-tuning"));
    #pragma atune endpermutation

    sortParallel(words);

    #pragma atune GAUGE mySortExecTime
    #pragma atune ENDBLOCK fillBlock

    countWords(words)
}

// Sorts string array
void sortParallel(List<string> words) {
    #pragma atune startblock sortBlock inside fillBlock

    IParallelSortingAlgorithm sortAlgo = new ParallelQuickSort();
    int depth = 1;
    #pragma atune setvar sortAlgo type generic
    values "new ParallelMergeSort(depth)", "new ParallelQuickSort()"
    scale nominal
    #pragma atune setvar depth type int
    values 1-4 scale ordinal
    depends sortAlgo='new ParallelMergeSort(depth)'

    sortAlgo.Run(words);

    #pragma atune endblock sortBlock
}

// Counts total characters of string array
int countCharacters(List<string> words) {
    #pragma atune startblock countBlock
    #pragma atune gauge myCountExecTime
    int numThreads = 2;
    #pragma atune setvar numThreads type int
    values 2-8 scale ordinal

    int total = countParallel(words, numThreads);

    #pragma atune gauge myCountExecTime
    return total;
    #pragma atune endblock countBlock
}

```

We designed Atune-IL as a pragma-based language for shared-memory multicore platforms. Direct support for message parsing architectures (such as MPI) is not included. All Atune-IL statements are preceded by a language-dependent pragma directive (such as `#pragma` for C++ and C# or `/*@` for Java) followed by the `atune` prefix. This approach offers two benefits: 1) tuning information

is separated from program code; 2) the instrumented program is executable even without auto-tuning, as pragmas and annotations are typically ignored by compilers.

Tuning Parameters. The `SETVAR` statement is used to mark a variable in the host language as tunable. Technically, it redefines an assignment of numeric or non-numeric parameters, and replaces the value by another one from a range specified as part of the `SETVAR` statement. Atune-IL generally assumes that the tuning parameter values are valid. The `generic` option allows the declaration of an arbitrary value to be assigned.

The `SETVAR` statement has a number of optional keywords to define additional specifications such as scale, weight, or context (see Table 1 for details).

An instrumented variable must be correctly declared in the host language and initialized with a default value. Atune-IL will modify this value at the point where the `SETVAR` instrumentation is located.

Parameter Dependencies. A conditional parameter dependency is defined by the optional `SETVAR` keyword `depends`. This causes a tunable parameter to be tuned only if the dependency condition evaluates to true. The condition may include more complex algebraic expressions.

Our program in Listing 1 contains the parameter `depth` that conditionally depends on parameter `sortAlgo`. The parameter `depth` is tuned only if merge sort is selected.

Permutation Regions. A set of host language statements can be marked to be permutable. The permutable statements are enclosed by a `STARTPERMUTATION` and `ENDPERMUTATION` statement, representing a permutation region. `NEXTELEM` delimits permutation elements; one permutation element may consist of several host language statements.

The example in Listing 1 shows a permutation region consisting of three permutation elements. Each element consists of a host language statement that adds an element to a list, so the whole region will generate a differently permuted list in different runs.

Measuring Points. Measuring points are inserted with the `GAUGE` statement, followed by a name to identify the measuring point. Atune-IL currently supports monitoring either execution times or memory consumption; the type of data to be collected is declared globally and uniformly for all measuring points. The developer is responsible for valid placements of measuring points.

The measuring points in Listing 1 are used to measure the execution time of two particular code segments (sorting and counting). For monitoring execution times, two consecutive measuring points with same name are interpreted as start and end time.

Blocks. Blocks are used to mark in a block-structured way the program sections that can be tuned independently (cf. Section 2.1). Such blocks run consecutively

Table 1. Atune-IL language features

<i>Statement</i>	<i>Description</i>
Defining Tuning Parameters	
SETVAR <identifier> type [int float bool string generic] values <value list>	Specifies a tuning parameter. type specifies the parameter's type. values specifies a list of numeric or non-numeric parameter values. Only feasible assignments are allowed.
scale? [nominal ordinal]	<i>Optional:</i> Specifies whether parameter is ordinal or nominal scaled.
default? <value>	<i>Optional:</i> Specifies the parameters default value.
context? [numthreads lb general]	<i>Optional:</i> Specifies the parameter's context to provide additional information to the auto-tuner.
weight? [0..10]	<i>Optional:</i> Specifies the parameter's weight regarding the overall application performance.
depends? <algebraic expr>	<i>Optional:</i> Defines a conditional dependency to another parameter. Algebraic constraints are supported.
inside? <block id>	<i>Optional:</i> Assigns the parameter logically to specified application block.
Defining Permutations Regions	
STARTPERMUTATION <identifier>	Opens a permutation region containing an arbitrary number of target language statements. The order of the statements can be permuted by an auto-tuner.
NEXTELEM	Separates the code elements in a permutation region.
ENDPERMUTATION	Closes a permutation region. Corresponding STARTPERMUTATION and ENDPERMUTATION statements must be in the same compound statement of the host programming language.
Defining Measuring Points	
GAUGE <identifier>	Specifies a measuring point, e.g., to measure time.
Defining Blocks	
STARTBLOCK <identifier>? inside? [<block id>]	Opens a block with an optional identifier. Optional STARTBLOCK keyword: Nests the block logically inside specified parent block. A block can have only one parent block.
ENDBLOCK	Closes a block. Corresponding STARTBLOCK and ENDBLOCK statements must be in the same compound statement of the host programming language. A lexically nested block must be declared entirely within its parent block.

in any of the application’s execution paths and their tuning parameters do not interfere with each other.

The code example in Listing 11 shows how to define blocks with Atune-IL. Basically, a block is enclosed by a `STARTBLOCK` and `ENDBLOCK` statement. Blocks have an optional name that can be referenced by other blocks.

Large parallel applications often contain nested parallel sections (cf. Section 2.1). To represent this structure, blocks support nesting as well – either lexically or logically. The latter requires the keyword `inside` after the `STARTBLOCK` definition to specify a parent block. A block can have one parent block, but an arbitrary number of child blocks.

Global parameters and measuring points are automatically bound to an implicit root block that wraps the entire program. Thus, each parameter and each measuring point belongs to a block (implicitly or explicitly).

In the example in Listing 11 `sortBlock` is logically nested within `fillBlock`, while `countBlock` is considered to be independent from the other blocks. According to our reduction concept mentioned in Section 2.1, the parameters `sortAlgo` and `depth` in `sortBlock` have to be tuned together with parameter `stringSearch` in `fillBlock`, as the order of the array elements influences sorting. This results in two separate search spaces that can be tuned one after another. These are: $\text{dom}(\text{fillOrder}) \times \text{dom}(\text{sortAlgo}) \times \text{dom}(\text{depth})$, and $\text{dom}(\text{numThreads})$ respectively. In addition, the dependency of `depth` on `sortAlgo` can be used to further prune the search space, because invalid combinations can be ignored.

It would be possible to obtain clues about independent program sections by code analysis. However, such an analysis may require additional program executions, or may deliver imprecise results. For these reasons, Atune-IL so far relies on explicit developer annotations.

4 Implementation of the Atune-IL Backend

We now discuss the principles of generating program variants. This is accomplished by the Atune-IL backend, consisting of the Atune-IL parser that pre-processes code to prune the search space, and a code generator. An auto-tuner connected to Atune-IL’s backend can obtain the values of all tuning parameters as well as feedback information coming from measuring points. Then, a new program variant is generated where tuning variables have new values assigned by the auto-tuner.

4.1 Generating Program Variants

The Atune-IL backend deduces the application’s block structure from `STARTBLOCK` and `ENDBLOCK` statements and analyzes dependencies of tuning parameters. It creates a data structure containing the corresponding meta-information and the current search space. If we connect an auto-tuner to the backend, the tuner can access this meta-information throughout the tuning process.

The generation of program variants is basically a source-to-source program transformation. Tuning parameter definitions are replaced by language-specific

code (e.g., `SETVAR` statements are replaced by assignments). Measuring points introduced by `GAUGE` statement are replaced by calls to language-specific monitoring libraries. After the Atune-IL backend has generated a program variant, it compiles the code and returns the autotuner an executable program.

4.2 Templates and Libraries for Language-Specific Code

The application of tuning parameter values as well as the calls to monitoring libraries require the generation of language-specific code. Atune-IL works with C#, C/C++, and Java, which are widely used general-purpose languages. For each language, there is a template file storing language specific code snippets, e.g., for variable assignment or calls to the monitoring library. Supporting a new host language and monitoring libraries thus becomes straightforward.

5 Experimental Results

In this section, we evaluate Atune-IL based on four case studies and applications.

MID. Our largest case study focuses on our parallelized version of Agilent’s MetaboliteID (MID) [15], a commercial application for biological data analysis. The program performs metabolite identification on mass spectrograms, which is a key method for testing new drugs. Metabolism is the set of chemical reactions taking place within cells of a living organism. MID compares mass spectrograms to identify the metabolites caused by a particular drug. The application executes a series of algorithms that identify and extract the metabolite candidates. The structure of MID provides nested parallelism on three levels that is exploited using pipeline, task, and data parallelism.

GrGen. GrGen is currently the fastest graph rewriting system [6]. For this case study, we parallelized GrGen and simulated the biological gene expression process on the E.coli DNA [7] as a benchmark. The model of the DNA results in an input graph representation consisting of more than 9 million graph elements. GrGen exploits task and data parallelism during search, partitioning, and rewriting of the graph.

BZip. The third case study deals with our parallelized version of the common BZip compression program [8]. BZip uses a combination of different techniques to compress data. The data is divided into fixed-sized blocks that are compressed independently. The blocks are processed by a pipeline of algorithms and stored in their original order in an output file. The size of the uncompressed input file we used for the experiments was 20 MB.

Sorting. The last case study focuses on parallel sorting. We implemented a program providing heap sort and parallel merge sort. To sort data items, the most appropriate algorithm can be chosen. Our sample array contained over 4 million elements to sort.

Table 2. Characteristics of the applications used in case studies

	<i>MID</i>	<i>GrGen</i>	<i>BZip</i>	<i>Sorting</i>
Host Language	C#	C#	C++	Java
Avg. Running Time (ms)	85,000	45,000	1,400	940
Approx. Size (LOC) ¹	150,000	100,000	5,500	500
Parallelism Types	Pipeline/ Task/Data	Task/ Data	Pipeline/ Task/Data	Task/ Data
# Identified Parallel Sections	6	3	2	2

Table 2 lists the key characteristics of each application. The programs selected for our case studies try to cover different characteristics of several application types and domains. In addition, they reveal common parallelism structures that are interesting for tuning. For each program, input data is chosen in a way that represents the program’s common usage in its application domain.

As an additional proof of concept, we developed a sample auto-tuner that worked with Atune-IL’s backend. The auto-tuner employs a common search algorithm, uses the search space information provided by Atune-IL, generates parameter configurations, starts the compiled program variants, and processes the performance results.

5.1 Results of the Case Studies

We instrumented the programs and let our sample auto-tuner iterate through the search space defined by Atune-IL. We performed all case studies on an Intel 8-Core machine².

Table 3 summarizes the results of all case studies. Although the programs may provide more tuning options, we have focused on the most promising parameters regarding their performance impact. The search space sizes result from the number of parameter configurations the auto-tuner has to check.

We now explain the parallel structure of the applications and how we used Atune-IL for instrumentation and search space reduction.

MID. MID has six nested parallel sections wrapped by Atune-IL blocks. The parent section represents a pipeline. Two of the pipeline stages have independent task parallel sections; one of them contains another data parallel section, while the other has two. In the task and data parallel sections we parameterized the number of threads and the choice of load balancing strategies. In addition, we introduced in the data parallel section parameters for block size and partition size; both depended on the load balancing parameters. Based on the nested structure, Atune-IL’s backend automatically created three separate, but smaller

¹ LOC without comments or blank lines.

² 2x Intel Xeon E5320 QuadCore CPU, 1.86 GHz/Core, 8 GB RAM.

Table 3. Experimental Results of Case Studies

	<i>Mid</i>	<i>GrGen</i>	<i>BZip</i>	<i>Sorting</i>
Atune-IL Instrumentation Statements				
# Explicit Blocks	5	3	2	0
# Tuning Parameters	13	8	2	2
# Parameter Dependencies	3	3	0	1
# Measuring Points	2	3	1	1
Reduction of Search Space				
Search Space Size w/o Atune-IL	24,576,000	4,849,206	279	30
Search Space Size with Atune-IL	1,600	962	40	15
Reduction	99%	99%	85%	50%
Performance Results of Sample Auto-tuner				
Best Obtained Speed-up	3.1	7.7	4.7	3.5
Worst Obtained Speed-up	1.6	1.8	0.7	1.3
Tuning Performance Gain ³	193%	427%	671%	269%

search spaces instead of a single large one. Considering the parameter dependencies as well, Atune-IL reduced the search space from initially 24,576,000 to 1,600 parameter configurations, that are manageable by common search algorithms. Our sample auto-tuner generates the necessary variants of the program. The best configuration obtains a speed-up of 3.1 compared to the sequential version. The moderate speed-up is caused by I/O operations, as the input data comes from hard disk and is too large for main memory.

GrGen. We wrapped three large parallel sections (search, partitioning, and rewriting of the graph) of GrGen by Atune-IL blocks. Due to Atune-IL’s capability to handle multiple measuring points within a program, we added a measuring point to each block to allow fine-grained execution time feedback. In several places, we parameterized the number of threads, the number of graph partitions, and the choice of partitioning and load balancing strategies. In addition, we defined a permutation region containing 30 permutable statements specifying the processing order of search rules. Three parameters had conditional dependencies. As the parallel sections are not marked as nested, the Atune-IL backend considered them as independent and therefore reduced the search space from 4,849,206 to 962 parameter combinations. The best configuration found by the sample auto-tuner brought a speed-up of 7.7.

BZip. Two performance-relevant parameters of BZip are the block size of input data (value range set to [100, ..., 900] bytes with step size of 100) and the number of threads for one particular algorithm in the processing pipeline (value range

³ The tuning performance gain represents the difference between the worst and the best configuration of the parallel program. We use it as an indicator for the impact of tuning.

set to $[2, \dots, 32]$ threads with step size of 1). As the block size does not influence the optimal number of threads and vice versa, we wrapped each parameter by a separate block. Thus, Atune-IL reduced the search space from $9 \cdot 31 = 279$ to $9 + 31 = 40$ parameter combinations. The best configuration achieved a speed-up of 4.7. Note that the worst configuration was even slower than the sequential version, which emphasizes the need for automated tuning.

Sorting. Our sorting application provides two tuning parameters, namely the choice of the sorting algorithm and the depth of the parallel merge sort that influences the degree of parallelism. As these parameters cannot be tuned separately, there is only one block. However, the parameter for the merge sort depth is relevant only if merge sort is selected. Therefore, we specified this as a conditional dependency, and the Atune-IL backend automatically cut the search space in half. The best configuration resulted in a speed-up of 3.5.

With our case studies we show 1) Atune-IL works with parallel applications differing in size, programming language, and application domain; 2) Atune-IL helps reduce the search space; 3) Atune-IL’s constructs are adequate for expressing necessary tuning information within a wide range of parallel applications.

6 Related Work

Specialized languages for auto-tuning have been previously investigated mainly in approaches for numeric applications, such as ATLAS [9], FFTW [10], or FIBER [11]. Below, we mention the most relevant approaches.

XLanguage [12] uses annotations to direct a C or C++ pre-processor to perform certain code transformations. The language focuses on the compact representation of program variants and omits concepts for search space reduction.

POET [13] embeds program code segments in external scripts. This approach is flexible, but the development of large applications is difficult, as even small programs require large POET scripts. By contrast, Atune-IL requires only one line of code to specify a tuning parameter or a measuring point.

SPIRAL [14] focuses on digital signal processing in general. A mathematical problem is coded in a domain-specific language and tested for performance. It works for sequential code only.

7 Conclusion

The increasing diversity of multicore platforms will make auto-tuning indispensable. Atune-IL supports auto-tuning by providing an efficient way to define tuning information within the source code. Key contributions of Atune-IL are the extended concepts for search space reduction, the support for measuring points, as well as the ability to generate program variants. In addition, portability is improved, since platform-specific performance optimization can now be easily handed over to an arbitrary auto-tuner. Of course, Atune-IL is in an early stage and can be improved. We are currently working on an approach to generate Atune-IL statements automatically for known types of coarse-grained parallel patterns.

Acknowledgments. We thank Agilent Technologies Inc. for providing the source code of MID as well as Agilent Technologies Foundation for financial support. We also appreciate the support of the excellence initiative in the University of Karlsruhe. Finally, we thank Thomas Karcher for his implementation of the Atune-IL parser [15].

References

1. Pankratius, V., et al.: Software Engineering For Multicore Systems: An Experience Report. In: Proceedings of 1st IWMSE, May 2008, pp. 53–60 (2008)
2. Asanovic, K., et al.: The Landscape of Parallel Computing Research: A View from Berkeley. Technical report, University of California, Berkeley (December 2006)
3. Tapus, C., et al.: Active Harmony: Towards Automated Performance Tuning. In: Proceedings of the Supercomputing Conference (November 2002)
4. Werner-Kytölä, O., Tichy, W.F.: Self-Tuning Parallelism. In: Proceedings of the 8th International Conference on High-Performance Computing and Networking, pp. 300–312 (2000)
5. Agilent Technologies: MassHunter MetaboliteID Software (2008), <http://www.chem.agilent.com>
6. Geiß, R., et al.: GrGen.NET. University of Karlsruhe, IPD Prof. Goos (2008), <http://www.info.uni-karlsruhe.de/software/grgen/>
7. Schimmel, J., et al.: Gene Expression with General Purpose Graph Rewriting Systems. In: Proceedings of the 8th GT-VMT Workshop (2009)
8. Pankratius, V., et al.: Parallelizing BZip2. A Case Study in Multicore Software Engineering. accepted September 2008 for IEEE Software (2009)
9. Whaley, R.C., et al.: Automated Empirical Optimizations of Software and the ATLAS Project. Journal of Parallel Computing 27, 3–35 (2001)
10. Frigo, M., Johnson, S.: FFTW: An Adaptive Software Architecture for the FFT. In: Proceedings of the International Conference on Acoustics, Speech and Signal Processing, May 1998, pp. 1381–1384 (1998)
11. Katagiri, T., et al.: FIBER: A Generalized Framework for Auto-tuning Software. In: Proceedings of the International Symposium on HPC, pp. 146–159 (2003)
12. Donadio, S., et al.: A Language for the Compact Representation of Multiple Program Versions. In: Proceedings of the 18th LCPC Workshop, pp. 136–151 (2006)
13. Yi, Q., et al.: POET: Parameterized Optimizations for Empirical Tuning. In: Proceedings of IPDPS, March 2007, pp. 1–8 (2007)
14. Püschel, M., et al.: SPIRAL: Code Generation for DSP Transforms. Proceedings of the IEEE 93, 232–275 (2005)
15. Karcher, T.: Eine Annotationssprache zur Automatisierbaren Konfiguration Parallel er Anwendungen. Master's thesis, University of Karlsruhe (August 2008)

Assigning Blame: Mapping Performance to High Level Parallel Programming Abstractions

Nick Rutar and Jeffrey K. Hollingsworth

Computer Science Department
University of Maryland
College Park, MD 20742, USA
`{rutar,hollings}@cs.umd.edu`

Abstract. Parallel programs are increasingly being written using programming frameworks and other environments that allow parallel constructs to be programmed with greater ease. The data structures used allow the modeling of complex mathematical structures like linear systems and partial differential equations using high-level programming abstractions. While this allows programmers to model complex systems in a more intuitive way, it also makes the debugging and profiling of these systems more difficult due to the complexity of mapping these high level abstractions down to the low level parallel programming constructs. This work discusses mapping mechanisms, called variable blame, for creating these mappings and using them to assist in the profiling and debugging of programs created using advanced parallel programming techniques. We also include an example of a prototype implementation of the system profiling three programs.

1 Introduction

As parallel systems become larger and more powerful, the problems that can be solved using these systems become larger and more complex. However, there is a divide between those software engineers and parallel language designers who know how to program for and utilize these systems and the people who actually have the problems that could use such systems. To help bridge this gap, recent years have seen more and more parallel frameworks and libraries [1, 2] arising to assist the application scientists in creating programs to utilize distributed systems. These environments have abstractions that hide away many of the lower level parallel language constructs and APIs that let developers program in terms of real world mathematical constructs.

All of the above points are positive for the scientific community, but they also bring about some interesting problems. An important issue introduced is profiling and debugging of these systems. There are many profiling tools that target parallel applications specifically. However, when these abstractions are introduced it becomes more and more difficult to diagnose runtime issues and debug them using conventional means. The higher level the abstractions, the harder it is to figure out the lower level constructs that map to them and subsequently discover where performance problems are occurring. This affects the

end user as well as the designers of the frameworks who are looking to improve the performance of their software.

Programs with multileveled abstractions introduce unique ways of approaching how to profile the program. For traditional profiling tools, measured variables (time, cache misses, floating point operations, etc.) are given in terms of easily delimited program elements (functions, basic blocks, source lines). Programs developed utilizing parallel frameworks can also be measured in such ways, but we believe that these traditional measurements can be improved upon. The interesting thing for many of these parallel frameworks is the representation of container objects in terms of things that many scientific end users relate to, e.g. Linear Systems, PDEs, Matrices. We believe that this can be utilized in a profiling environment by taking the abstractions a step further and representing performance data in terms of the abstractions, mainly the instantiations of these abstractions in the form of program variables. The unique feature of our system is its ability to automatically combine and map complex internal data structures (such as sparse matrices and non-uniform grids) to higher level concepts.

The core of mapping performance data to the variables rests with the idea of variable “blame.” Variable blame is defined as a measurement of performance data associated with the explicit and implicit data flow used to determine the value of a particular variable at various levels of an abstraction’s hierarchy. The performance data metric is chosen by the end user based on what profiling information they are interested in, which is in turn based on the available measurable hardware counters for the system. For complex structures or classes that have multiple member variables ranging from other complex types or primitives, it is the aggregation of all the blame from its respective components. Blame is calculated at the lowest level, where data movement is made in the form of individual assignment and arithmetic operations, and then bubbled up the hierarchy of the abstraction.

2 Calculating Blame

The calculation of blame for a given variable is a multistep process utilizing both static and runtime information. When possible, information is gathered statically once to decrease the runtime footprint. The runtime operations are based primarily on information gathered through sampling utilizing hardware counters. The process for calculating blame is discussed in this section and is displayed in Figure 11.

2.1 Data Flow Relationships

The building blocks for variable blame lie in the data flow relationships between variables in a program. Put simply, at the end of a code region we want to examine what variable ultimately contains the product of all the work that went into producing a certain value. There are two types of data flow relationships we are interested in, explicit and implicit.

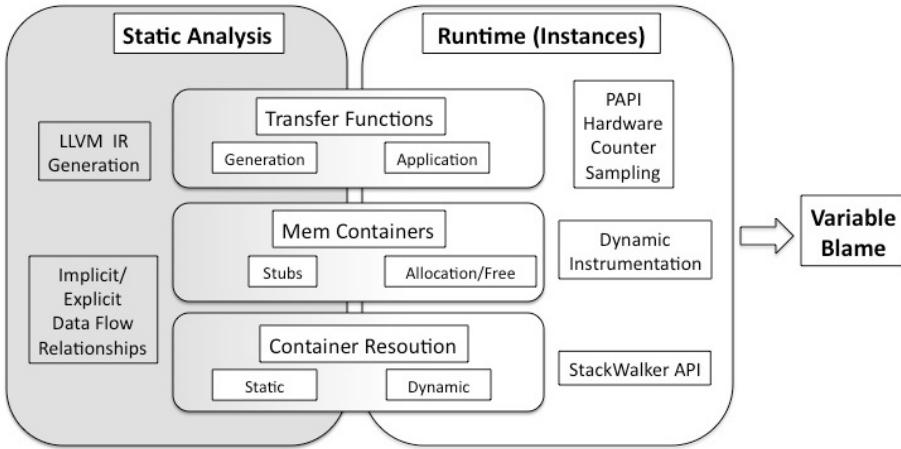


Fig. 1. Blame mapping for main function

Explicit transfers occur on data writes within a program. For example, consider the C snippet below:

```
int a, b, c;
a = 7;
b = 8;
c = a + b;
```

The blame for this snippet would be assigned to variable *c*. This is because the values of *a* and *b* are calculated directly for the purpose of having their sum stored in *c*. The variable *c* may then be used in another computation and will subsequently be included in the blame set of the variable that uses it (along with *a* and *b*).

Implicit transfer is more complicated and subjective than explicit transfer. It primarily concerns variables that are assigned a value that is never directly assigned to another variable. This occurs often in variables that involve control flow. For example, a loop index is incremented for every iteration of the loop and used in a comparison operation but is never actually assigned in some computation (besides incrementing itself). This is also true in flags for switch statements and standard conditional statements. In these cases, all of the variables implicitly affected by these variables (statements within loop body, conditional branch) will have an implicit relationship with these variables.

Both explicit and implicit relationships are computed statically and information is stored per function. We accomplish this by analyzing an intermediate, three address, representation of the program. We describe memory operations in Section 2.3. For explicit relationships, we build a graph based on the data flow between the variables. For implicit relationships, we use the control flow graph and dominator tree generated to infer implicit relationships for each basic block. All variables within those basic blocks then have a relationship to the implicit

variables responsible for the control flow that resulted in the blocks present in that control flow path. For both implicit and explicit relationships, after the calculations are performed on the intermediate format, a mapping is used to relate data back to the original source code.

2.2 Transfer Functions

The data flow relationships are all recorded at the function level and calculated through intraprocedural analysis. For interprocedural analysis, we need a mechanism to communicate the blame between functions. We utilize *transfer functions* for this step. When looking at explicit and implicit blame, we utilize a form of escape analysis [3] to determine what variables, which we deem exit variables, are live outside of the scope of the function. These could be parameters passed into the function, global variables, or return values. All explicit and implicit blame for each function is represented in terms of these exit variables during static (pre-execution) analysis. During runtime, a transfer function is used to resolve the caller side parameters and return containers to the appropriate callee side exit variables.

When source code is not available, manual intervention is needed. For these cases, transfer functions can be created based on knowledge about a procedure's functionality. When faced with a complete lack of knowledge about a function (no source or documentation) a heuristic is used to divide up the blame between the parameters and return values from these functions.

For common library routines such as MPI, we include predefined blame assignments based on the prototype of the functions. For example, if we see a program with a call to MPI_Bcast without doing any additional analysis we can attribute the blame for that function to the variable passed in as the first parameter, the data that is being broadcast.

2.3 Mem-Containers

Up to this point, the discussion has not considered operations involving memory, whether stack or heap allocated. We represent these operations in our mappings with "mem-containers." A mem-container is an abstraction representing a unique contiguous region of memory for the scope of time the memory is available to be accessed by the program. In the case of heap objects, this would be the period of time between allocation and deallocation. For global variables, the period of time would be the length of the program. For structures, classes, and arrays using memory allocated on the stack, the supporting mem-container is active while the function that pushed the data to the stack is still represented on the call stack. It should be noted that mem-containers are not the final blame containers that are presented to the user. There needs to be an additional mapping that takes place to associate the mem-containers with the program variables they represent. Abstractions within frameworks for items such as matrices and vectors may have multiple mem-containers associated with them.

Like transfer functions, static analysis is used to reduce runtime data collection and analysis. For mem-containers, we can determine points in the program where

memory allocation and deallocation takes place. At these places in the code, we create stubs signifying that an allocation can be expected at these points when the program is run. At runtime, by instrumenting allocation routines like *malloc*, we gather additional information such as the location in memory the allocation took place and the size of the allocation. Furthermore, we perform a stack walk at each allocation point to determine the call path that led to the allocation. This path information is used to match the allocation to the stubs we had generated through static analysis.

2.4 Container Resolution

Container resolution refers to the resolution of blame within complex data types. For instance, a structure or class may have multiple fields which acquire blame through the course of the program. Some of these fields may themselves be classes. Container resolution is simply the bubbling up of blame until it reaches the top most container type. Much of this resolution can be taken care of statically, though there could be cases where a field may be referenced through a set of pointers where runtime information will be needed to fully attribute blame to the proper container.

2.5 Instance Generation

An instance is an abstraction that represents the operations that occur at a specific sampling point. Since each sample generates a unique instance, the instance will carry with it the blame of the metric that caused the program to generate the sample. The metric is chosen by the user based on what aspect of the program they would like to measure, and typically are of the set of hardware counters available on a given platform. Instances are then mapped up to either mem-containers or program variables for aggregation of blame. In the case that the access pertained to a primitive within the code, the instance will map to a specific program variable and no record of the instance will need to be maintained. In the case the sample occurred during a memory operation, more exhaustive bookkeeping about the instance is maintained and an upmapping is created to its corresponding mem-container. Within each instance are also identifiers that describe which node in a distributed system and/or thread these accesses came from.

This sampling can be done not only at specific time intervals, but can also be based on program events. For example, threshold driver overflow interrupts from hardware counters can be used. The user is allowed to choose what metric(i.e. cache misses, floating point operations) they want to sample and how often the interrupt will be triggered. When the interrupt occurs, our system uses a handler that records the program counter and state of the machine when the overflow occurs. We use this interrupt as an opportunity to do a stack walk at the point of interrupt. This allows us to utilize the transfer functions in a context sensitive matter.

2.6 Final Variable Blame

The final blame for a given variable is presented to the user at “blame points” throughout the program. The majority of blame points are automatically identified through static analysis as places within the program where blame can not be mapped up any further to another variable, the most obvious of these functions being the *main* function. A blame point can also be any point in the program explicitly designated by the user to be a point of interest or that falls into a set of criteria that would make it interesting to examine. One such set of criteria could be having a function with variables that contain more than a certain percentage of the total blame for the program. It should also be noted that although blame propagates up the call stack, not all blame will necessarily make it back to the original *main* function. An example of this, as well as detailing multiple blame points, is shown in Section 3.3.

3 Experimental Results

To implement the concepts introduced in Section 2 we used various components. To generate the intermediate format used to calculate the implicit and explicit relationships, we used LLVM[4]. To perform the stackwalking that assisted in creating a context sensitive representation of the mem-containers and instances, we used the Stackwalker API[5]. Finally, for generating instances through sampling we used PAPI[6]. We also utilized PAPI for accessing hardware counter information.

To show some of the capabilities that our mapping offers that differ from traditional techniques, we have chosen three programs that directly exhibit properties that would be found in large parallel programming abstractions. For all three programs, the blame metric concerns cycles spent with the sampling triggered every predetermined number of cycles. Therefore, for each sampling point (instance), whatever variable gets the blame for that instance is essentially responsible for the cycles that were used between measured samples. For these experiments, we present the absolute blame numbers that are matched one to one with the samples taken while profiling the program. We also present the percentage of the program cycles that were used in the calculation of the variable according to our blame mappings.

It should be noted that although we used cycles for the blame metric for these tests, once the mappings are in place any measurable metric on the system can be applied. In the case of parallel programs, we can utilize metrics such as blocking time and bytes transferred in cases where MPI calls contribute to the blame.

3.1 FFP_SPARSE

One of the test programs we examined was FFP_SPARSE[7], a small open source C++ program that uses sparse matrices and a triangle mesh to solve a form

of Poisson's equation. It consists of approximately 6,700 lines of code and 63 functions. Although this program is sequential, the problem space and data structures utilized make it an attractive case study.

We ran the FFP_SPARSE program and recorded 101 samples which are the basis of the mappings discussed in this section. After removal of the debugging output, the only blame point for this program is the main function, with the program culminating in the output of the final solution vector.

This program does not have complex data structures to represent vectors and matrices, but the variable names for the primitive arrays map nicely to their mathematical counterparts in many cases. Table \textcolor{red}{II} shows the blame mappings for the variables in *main*. The “Base Data Centric” column represents explicit memory operations, meaning that the sampling was taken when an assignment was occurring for these arrays. “Blame” refers to the number of samples in which blame was assigned to those variables (whether directly to the variable or the mem-containers that mapped to that variable).

One thing that stands out from this sampling is the lack of sample points (only two) where an explicit write was taking place to the arrays present at the top scope of *main*. This number includes any writes to these memory locations under all aliases as many of these arrays are passed as parameters throughout the program. If one were looking to examine which variables were the most important to the program based solely on this information, which would be the case for standard profiling systems, it would be hard to get any usable information. In a dense matrix, there may be many more reads and writes to actual memory locations tied to the defined variable for the matrix. However, in sparse matrix implementations, many of the computations take place behind layers of abstraction between the defined variable and where the work is actually taking place. When blame mapping is introduced we get a clearer picture of what the program is trying to accomplish. The solution vector and the coefficient matrix are the clear recipients for most of the blame of the program.

Table 1. Variables and their blame for run of FFP_SPARSE

Name	Type	Description	Base Data Centric	Blame(%)
<i>node_u</i>	double *	Solution Vector	0	35(34.7)
<i>a</i>	double *	Coefficient Matrix	0	24.5(24.3)
<i>ia</i>	int *	Non-zero row indices of <i>a</i>	1	5(5.0)
<i>ja</i>	int *	Non-zero column indices of <i>a</i>	1	5(5.0)
<i>element_neighbor</i>	int *	Estimate of non-zeroes	0	10(9.9)
<i>node_boundary</i>	bool *	Bool vector for boundary	0	9(8.9)
<i>f</i>	double *	Right hand side Vector	0	3.5(3.5)
Other	-	-	0	9(8.9)
Total	-	-	2	101(100)

Table 2. Variables and their blame for run of the QUAD_MPI

Name	Type	MPI Call	Blame (per Node)				
			N1(%)	N2(%)	N3(%)	N4(%)	Total(%)
<i>dim_num</i>	int	MPI_Bcast	27(27.2)	90(95.7)	97(84.3)	102(94.4)	316(76.0)
<i>quad</i>	double	MPI_Reduce	19(19.2)	1(1.1)	5(4.3)	5(4.6)	30(7.2)
<i>task_proc</i>	int	MPI_Send	15(15.2)	-	-	-	15(3.6)
<i>w</i>	double*	-	9(9.1)	-	-	-	9(2.1)
<i>point_num_proc</i>	int	MPI_Recv	-	1(1.1)	7(6.1)	-	8(1.9)
<i>x_proc</i>	double*	MPI_Recv	-	2(2.1)	5(4.3)	-	6(1.4)
Other	-	-	3(3.0)	-	-	-	3(0.7)
Output	-	-	6(6.1)	-	1(0.9)	1(0.9)	8 (1.9)
Total		-	99(100)	94(100)	115(100)	108(100)	416(100)

3.2 QUAD_MPI

QUAD_MPI[8] is a C++ program which uses MPI to approximate a multidimensional integral using a quadrature rule in parallel. While the previous program illustrated how a sparse data structure can be better profiled using variable blame, this program helps to illustrate how some MPI operations will be modeled. It is approximately 2000 lines of code and consists of 18 functions.

We ran the QUAD_MPI program on four Red Hat Linux nodes using OpenMPI 1.2.8 and recorded a range of 94-108 samples between the four nodes. As discussed in Section 2.2, all calls to MPI functions were handled by assigning blame to certain parameters based on the prototypes of the MPI programs utilized. The program exits after printing out the solution, represented by the variable *quad*.

The results for the run are shown in Table 2. The variables are listed in descending order based on the total amount of blame assigned across all nodes. For each variable, it is shown whether an MPI operation was a contributing factor, but not necessarily the only source, of the blame. This program spends a majority of its time reading in the data files and processing MPI calls with the computation between those calls minimal. The variable with the most blame, *dim_num*, is due to program input from the master node at the beginning of the program, which causes the three other nodes to create an implicit barrier. The second highest blame count goes to *quad*, which is the variable that holds the output for the program so a high number is to be expected.

3.3 HPL

HPL[9] is a C program that solves a linear system in double precision on distributed systems. It is an implementation of the “High Performance Computing Linpack Benchmark.” Unlike FFP_SPARSE, the operations are done on dense matrices. HPL offers a variety of attractive features as a test program for blame mapping. It utilizes MPI and BLAS calls and has wrappers for the majority of

the functions from both libraries. While the previous two programs were smaller programs, HPL has approximately 18,000 lines of code over 149 source files.

HPL is also interesting to examine because it is similar to many parallel frameworks in that MPI communication is completely hidden from the user. This means tracing MPI bottlenecks using traditional profiling techniques may technically give you information about where the bottleneck is occurring. However, that information may be useless because the MPI operations are buried deeply enough in complex data structures that knowing how these bottlenecks affect variables at the top levels of the program is difficult to discover.

We ran the HPL program on 32 Red Hat Linux nodes connected via Myrinet using OpenMPI 1.2.8 and recorded a range of 149-159 samples between the nodes. The results for the run are shown in Table 3. This program differs from the other two in that we have multiple blame points. Two of these blame points would be explicitly generated. The other two are user selected and contain variables (*A* and *PANEL*) that have a large amount of blame associated with them. The *main* function serves primarily to read in program specifications and iterate through the tests, which have their own output. For this reason, only the computation associated with producing the computation grid is actually attributed to it while one its called functions (*HPL_pdtest*) has a much larger stake of the total blame.

In terms of the variables themselves, two different blame points, *mat* and *A* (which is a pointer to *mat*), are assigned the majority of the blame. This is

Table 3. Variables and their blame at various blame points for run of HPL

Name	Type	Blame over 32 Nodes	
		Node Mean(Total %)	Node St. Dev.
All Instances	-	154.7(100)	2.7

main

<i>grid</i>	HPL_T_grid	2.2(1.4)	0.4
-------------	------------	----------	-----

main→HPL_pdtest

<i>mat</i>	HPL_T_pmat	139.3(90.0)	2.8
<i>Anorm1</i>	double	1.4(0.9)	0.8
<i>AnormI</i>	double	1.1(0.7)	1.0
<i>XnormI</i>	double	0.5(0.3)	0.7
<i>Xnorm1</i>	double	0.2(0.1)	0.4

main→HPL_pdtest→HPL_pdgesv

<i>A</i>	HPL_T_pmat *	136.6(88.3)	2.9
----------	--------------	-------------	-----

main→HPL_pdtest→HPL_pdgesv→HPL_pdgesv0

<i>PANEL</i> → <i>L2</i>	double*	112.8(72.9)	8.5
<i>PANEL</i> → <i>A</i>	double *	12.8(8.3)	3.8
<i>PANEL</i> → <i>U</i>	double *	10.2(6.6)	5.2

intuitive since A contains a pointer to the local data for the matrix being solved as well as the solution vector and some book keeping variables. Going deeper down the call trace, we find the variable *PANEL* which is a structure with three fields that carry a large portion of the blame of the program. These variables are the storage containers for the computations in the LU factorization and through container resolution it can be presented to the user as the single variable *PANEL* or as separate fields. The blame is assigned locally to these three variables and through a transfer function is then assigned to A in the caller function.

All the variables in Table 3 had contributions to their blame total from MPI operations that occurred within HPL wrappers far down the call stack. HPL is a well established benchmark so one can see that the decomposition is very efficiently done with each variable being calculated almost equally across the processors. An interesting caveat to this balance can be found when looking at the fields within $PANEL(L2, A, U)$, which have many MPI calls contributing to the blame for these variables. For these nodes, there is some imbalance when calculating some of these individual components but whatever imbalance occurs immediately disappears when these variables have their blame absorbed by A one level up on the stack. This is an interesting result, as standard profiling programs might be concerned with the apparent bottlenecks occurring for these lower operations when in fact the calculation of the solution vector (a field in A) is calculated with excellent load balance between the nodes.

4 Related Work

The main area of related work deals with creating mappings in parallel programs at different levels of abstraction. Irvin introduces concepts involving mapping between levels of abstraction in parallel programs with his NV model [10] as utilized by the ParaMap [11] tool. The Semantic Entries, Attributes, and Associations(SEAA) [12], a followup to the NV model, addresses some of the limitations of the NV model and adds some additional features. The SEAA mappings are utilized in the TAU performance tool. [13]. The primary difference between our mapping and these is the way the performance data is collected and transferred. In NV and SEAA, regions of code are measured and mapped up different levels of abstraction with these regions eventually being mapped to variables in some cases. Our approach maps to variables at every level of abstraction and uses data flow operations as the primary mechanism for transferring performance data. Using data flow allows us to push much of the computation to static analysis, with runtime information supplementing that data whereas NV and SEAA focus primarily on runtime information. Finally, we use sampling as our primary technique for generating data versus delimited instrumentation of a code region and measurements taken of that region.

Profiling tools that utilize sampling are the most similar to our approach. These tools include prof [14], gprof [15], DCPI [16], HPCToolkit [17], and Speedshop [18]. Similar instrumentation based profiling tools include TAU [13] and SvPablo [19]. Like the mappings discussed above, the primary focus of these tools is code regions and base data structures.

Dataflow analysis is utilized in many areas. Guo et al. [20] dynamically records data flow through instrumentation at the instruction level to determine abstract types. This is similar to the explicit data flow relationships we use in our blame analysis. Other work dealing with information flow in control flow statements is similar to the implicit data flow relationships we use [21][22].

5 Conclusions and Future Work

In this paper, we have outlined a style of variable mapping called blame mapping. Blame mapping looks at the explicit assignments made to a variable directly, but also concerns itself with all of the computation that went into any variable that is eventually assigned to that variable. The computation of blame mapping takes place partly in an intraprocedural manner to determine the exit variables for a function and how much blame each exit variable is assigned. These exit variables are then combined into a transfer function so an interprocedural analysis can be constructed. This interprocedural analysis is combined with runtime information obtained by sampling to create a repository of information from a program run that can be used for performance analysis or debugging.

The target applications for these types of mappings are large parallel programs with many levels of abstraction, specifically large scientific frameworks. Their abstractions are often tied to mathematical constructs so a representation of performance data in terms of variables may simplify the analysis process. Furthermore, they are often large, long running parallel programs so the less intrusive footprint introduced by sampling is desirable.

Our current experiments have involved smaller programs that exhibit similar properties to those larger programs described above. The primary focus of our future work will involve applying our mapping to these complex abstractions and the larger programs that utilize them. This paper has included profiling data attributed to data structures that is intuitive to where the blame should lie and serves as a sanity check that this style of mapping can produce accurate results. In more complex data structures, these mappings will not be as intuitive. For our future work, we will provide detailed case studies with these large frameworks and how to utilize the mapping information. Furthermore, we will provide a quantitative comparison between results given from blame mapping and traditional profiling tools.

References

1. POOMA, <http://acts.nersc.gov/pooma/>
2. Balay, S., Buschelman, K., Gropp, W.D., Kaushik, D., Knepley, M.G., McInnes, L.C., Smith, B.F., Zhang, H.: PETSc Web page (2001), <http://www.mcs.anl.gov/petsc>
3. Deutsch, A.: On the complexity of escape analysis. In: POPL 1997: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 358–371. ACM, New York (1997)

4. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization, CGO 2004 (2004)
5. Univ. of Maryland, Univ. of Wisconsin: StackWalker API Manual. 0.6b edn. (2007)
6. Browne, S., Dongarra, J., Garner, N., London, K., Mucci, P.: A scalable cross-platform infrastructure for application performance tuning using hardware counters, pp. 65–65 (2000)
7. FFP_SPARSE, http://people.scs.fsu.edu/~burkardt/cpp_src/ffp_sparse/
8. QUAD, http://people.sc.fsu.edu/~burkardt/c_src/quad_mpi/
9. HPL, <http://www.netlib.org/benchmark/hpl/>
10. Irvin, R.B.: Performance Measurement Tools for High-Level Parallel Programming Languages. PhD thesis, University of Wisconsin-Madison (1995)
11. Irvin, R.B., Miller, B.P.: Mapping performance data for high-level and data views of parallel program performance. In: International Conference on Supercomputing, pp. 69–77 (1996)
12. Shende, S.: The Role of Instrumentation and Mapping in Performance Measurement. PhD thesis, University of Oregon (2001)
13. Shende, S.S., Malony, A.D.: The tau parallel performance system. Int. J. High Perform. Comput. Appl. 20(2), 287–311 (2006)
14. Graham, S.L., Kessler, P.B., McKusick, M.K.: An execution profiler for modular programs. Softw., Pract. Exper. 13(8), 671–685 (1983)
15. Graham, S.L., Kessler, P.B., McKusick, M.K.: gprof: a call graph execution profiler. In: SIGPLAN Symposium on Compiler Construction, pp. 120–126 (1982)
16. Anderson, J., Berc, L., Dean, J., Ghemawat, S., Henzinger, M., Leung, S., Sites, D., Vandevoorde, M., Waldspurger, C., Weihl, W.: Continuous profiling: Where have all the cycles gone (1997)
17. Mellor-Crummey, J.M., Fowler, R.J., Whalley, D.B.: Tools for application-oriented performance tuning. In: International Conference on Supercomputing, pp. 154–165 (2001)
18. SGI Technical Publications: SpeedShop User’s Guide
19. De Rose, L., Zhang, Y., Reed, D.A.: SvPablo: A multi-language performance analysis system. In: Puigjaner, R., Savino, N.N., Serra, B. (eds.) TOOLS 1998. LNCS, vol. 1469, pp. 352–355. Springer, Heidelberg (1998)
20. Guo, P.J., Perkins, J.H., McCamant, S., Ernst, M.D.: Dynamic inference of abstract types. In: ISSTA 2006: Proceedings of the 2006 international symposium on Software testing and analysis, pp. 255–265. ACM, New York (2006)
21. McCamant, S., Ernst, M.D.: Quantitative information flow as network flow capacity. In: PLDI 2008, Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 9–11, pp. 193–205 (2008)
22. Volpano, D., Irvine, C., Smith, G.: A sound type system for secure flow analysis. J. Comput. Secur. 4(2-3), 167–187 (1996)

A Holistic Approach towards Automated Performance Analysis and Tuning[∗]

Guogjing Cong, I-Hsin Chung, Huifang Wen, David Klepacki, Hiroki Murata,
Yasushi Negishi, and Takao Moriyama

IBM Research

Abstract. High productivity to the end user is critical in harnessing the power of high performance computing systems to solve science and engineering problems. It is a challenge to bridge the gap between the hardware complexity and the software limitations. Despite significant progress in language, compiler, and performance tools, tuning an application remains largely a manual task, and is done mostly by experts. In this paper we propose a holistic approach towards automated performance analysis and tuning that we expect to greatly improve the productivity of performance debugging. Our approach seeks to build a framework that facilitates the combination of expert knowledge, compiler techniques, and performance research for performance diagnosis and solution discovery. With our framework, once a diagnosis and tuning strategy has been developed, it can be stored in an open and extensible database and thus be reused in the future. We demonstrate the effectiveness of our approach through the automated performance analysis and tuning of two scientific applications. We show that the tuning process is highly automated, and the performance improvement is significant.

1 Introduction

Developments in high performance computing (HPC) have primarily been driven so far by bigger numbers of floating-point operations per second (FLOPS). Complex HPC systems pose a major challenge to end users to effectively harness the computing capability. It can take tremendous efforts of a user to develop an application and map it onto the current supercomputers. The US department of advanced research projects agency (DARPA) sponsored the high productivity computing system initiative [7] to develop technologies that help bridge the productivity gap. Improving productivity and maintaining high performance involve multiple areas of computer science research. Despite significant progress, deploying an application to complex architectures for high performance remains a highly manual process, and demands expertise possessed by few engineers.

A typical tuning life cycle is as follows. When the performance of an application is below expectation, a user is faced with the task of observing the behavior,

[∗] This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.

formulating hypothesis, and conducting validation tests. First the application is instrumented for performance data collection. Crude hypothesis can be formed based on the data, and observing for minute details (also called tracing) to refine or validate follows. Assuming trace data (oftentimes they are of a huge amount) can be efficiently manipulated, a user has to correlate the runtime behavior with the program characteristics. If a mismatch between the application and the architecture/system is detected, the user usually needs to trace back to the source program to find where the mismatch is introduced. Familiarity with the internals of the compiler is a must since a program goes through transformations by a compiler. The mismatch can be introduced by the compiler (not implementing the best transformations due to its constraints), or by the source program (e.g., implementing a poorly performing algorithm). Once the cause for the performance problem is identified, the user chooses the appropriate ones from her repertoire of optimizations for implementation. In doing so she needs to make sure the transformation does not violate program constraints, and conserve other desired properties such as portability.

Thus performance diagnosis requires in-depth knowledge of algorithm, architecture, compiler, and runtime behavior, and involves collecting, filtering, searching, and interpreting performance data. Tuning further requires coordinating the intricate interactions between multiple components of a complex system. Performance analysis and tuning remain challenging and time consuming even for experienced users. In our study we improve the tuning productivity by providing software services that help the automation of the process. Automated performance optimization has been studied at limited scales (as opposed to whole application or system scale) under different contexts. Here we give brief summary of related work.

An optimizing compiler is one of the most important auto-tuners. Profile-guided compilers (e.g., see [3]) find even further optimization opportunities in a program by utilizing both static and runtime information. Powerful as it is, a compiler does not tune well programs that heavily utilize specialized *primitives* that are largely outside a compiler's control. Some examples of these primitives include MPI communication, I/O operation, and pre-developed library routines. A compiler is usually not able to conduct transformations on the algorithms. It also lacks domain knowledge that is crucial to tunings dependent on the inputs. Even within standard loop transformations, the compile time constraint oftentimes does not allow a compiler to search for the best parameters or combinations.

Auto-tuning libraries (e.g., see [16,14]) are able to tune themselves (e.g., search for appropriate parameter values) for a given architecture configuration. As the tuning is for a predefined set of programs (usually much smaller than a regular application) with relatively limited transformations, accurate modeling and intelligent searching are the main effective techniques. These techniques alone in general will not tune an application.

Performance tools (e.g., see [11,10,12,13]) traditionally facilitate performance data collection and presentation. They aim to provide clues to trained experts

about possible performance problems. They in general do not explicitly point out the problem itself, and require a lot of efforts to digest the performance data.

In this paper we present our approach towards automated performance analysis and tuning. We work with experts to observe the way they tune their applications, and attempt to automate the procedure by mining their knowledge. We developed a framework that actively searches for known performance patterns instead of passively recording all the tracing information. Such a framework is designed to be open and extensible to accommodate new performance patterns. We also provide a mechanism to collect, compare, and correlate performance data from different aspects of the system for performance diagnosis. The framework also attempts to mitigate performance problems by suggesting and implementing solutions. Our approach attempts to unify performance tools, compiler, and expert knowledge for automated performance tuning.

The remainder of this paper is organized as follows: Section 2 describes our strategy for the development of automatic performance optimization tools. Section 3 describes our approach for incorporating multiple performance tools and compiler analysis in performance problem diagnosis. Section 4 presents automated solution proposal. Section 5 presents our case study. We present concluding remarks and future work in Section 6.

2 Performance Optimization Strategy

Working with performance experts, we propose a holistic approach towards performance analysis and tuning. We provide assistance to the user throughout the tuning process, that is, our framework provides services for performance data collection, bottleneck identification, solution discovery and implementation, and iteration of the tuning process. The framework strives to give definite diagnosis to a problem so that the user is no longer left with a sea of perplexing data. Performance tools, compiler, and expert knowledge are key components of our system. Integrating these components improves the tuning efficiency by providing means to compare and correlate performance data from different dimensions, and to propose and implement solutions.

Our work focuses on three levels of support to boost productivity. The first is to provide easy access to a wide array of information from static analysis, runtime behavior, algorithm property, architecture feature, and expert domain knowledge. Based upon such information, a mechanism to compare and correlate performance metrics from different aspects (e.g., computation, memory, communication, I/O) is developed, and helps accurately pinpoint the cause of performance problems. The ultimate goal is to automate tuning as an expert would. It involves proposing solutions and implementing them. Granted that currently (even in the near future) we do not foresee our framework to be able to generate knowledge itself, it can be instructed to follow certain tuning routines of an expert. Mining as much knowledge as possible from the experts liberate them from the repetitive tasks and is expected to greatly improve the productivity of regular users.

Our methodology can be summarized as follows. We collect the cause of performance problems from literature and performance experts, and store them as patterns defined on performance metrics. Our framework inspects and instruments the application, and actively searches for known patterns in the pattern database. Once a pattern is discovered, we claim that the corresponding bottleneck is found, and the framework consults the knowledge database for possible solutions. Solutions are evaluated, and implemented if desired by the user. There are two aspects of our work. One is infrastructure support. That is, we develop the necessary facilities for the automation process. The other is to populate the framework with a rich collection of bottleneck and solution definitions. We realize the sheer number of performance problems can easily overwhelm any study group. We make the framework open and extensible, and distill common utilities that help expert users expand the databases.

3 Bottleneck Discovery

A bottleneck is the part of a system that limits the performance. Achieving maximum performance can probably be formulated as a gigantic combinatorial optimization problem. Yet the sheer complexity determines that oftentimes we have to resort to heuristics for sub-optimal solutions. In practice, bottleneck discovery requires knowledge about application domain, compiler, software system, and architecture, and is traditionally done by a small group of experts. A mechanism to mine the expert knowledge is necessary to automate the tuning process.

The mining act is not trivial as the wisdom is often expressed in fuzzy terms. Formalizing the knowledge takes effort as shown by the following example. MPI derived data types have been proposed to improve the flexibility and performance of transferring non-contiguous data between processors. Not all programmers are fluent with derived data types, and they still compact data explicitly into a contiguous buffer before communication. This practice results in additional copying overhead, and can seriously degrade performance. The tuning guideline from the expert is clear. Simply stated, it asks to get rid of the redundant data packing by using derived data types. Yet it is not straightforward for a machine to detect the packing behavior, especially when the packing is done in a different function than where the communication occurs. Moreover, the packing itself can be complicated, and involves several loop regions. To identify the buffer being sent is simple (trivially through MPI tracing), but confirming the fact that data was being packed is hard. We formalized a scheme that relies on runtime memory access analysis (intercepting loads/stores to the buffer at run time), and flow analysis (through static analysis) to discover the behavior. This example emphasizes the need to integrate tools and compiler for bottleneck detection.

A bottleneck in our framework is a rule (pattern) defined on a set of metrics. The rule is expressed in a mini-language, and most current rules are logic

expressions. The design goal of the language is to be flexible and expressive enough to cover most frequent problem patterns. As more knowledge about bottlenecks is developed, new language features might be added. Rule definitions are acquired from literature and expert knowledge. The rule database is designed to be open for expansion and customization.

In our design a performance metric is any quantifiable aspect about or related to application performance. Examples include the number of pipeline stalls for a given loop, the number of prefetchable streams, the number of packets sent from a certain processor, the size of physical memory, and whether loops have been tiled by the compiler. The bottleneck rule provides a way to compare and correlate metrics from multiple sources and dimensions, and helps the accurate diagnosis of the problem. Having a large collection of metrics helps introducing new rules.

3.1 Metrics from Existing Performance Tools

Each existing performance tool is specialized in collecting certain performance metrics. These metrics can be used in building meaningful bottleneck rules through a mechanism our framework provides for metric import.

In our current implementation, the framework is able to collect many metrics through the IBM high performance computing toolkit (IHPCT) [15]. IHPCT contains a profiling tool [2], a hardware performance monitor (HPM), a simulation guided memory analyzer (SiGMA) [5], an MPI profiling and tracing tool, an OpenMP tracing tool, and a modular I/O tool. Each of these components evaluates and/or measures certain performance aspect of the application, and the wealth of performance data collected serve as metrics for bottleneck definitions. For example, HPM alone collects up to hundreds of metrics about various hardware events.

Metrics from different tools can co-exist, and we also experiment with collecting metrics through TAU [10] and Scalasca [6]. Table 1 shows some sample metrics collected by existing tools.

Combining the analysis of multiple tools can be simply achieved by defining rules that use metrics collected by them. For example, the following rule points to a potential pipeline stalling problem caused by costly divide operations in a loop.

Table 1. Example metrics collected by different performance analysis programs

metric name	description	collected by
PM_INST_CMPL	instruction completed	HPM
L1_miss_rate	L1 miss rate	HPM
Avg_msg_size	average message size	MPI profiler
Thread_imbalance	thread work load imbalance	Open MP profiler
#prefetches	number of prefetched cache lines	SiGMA
mpi_latesender	Time a receiving process is waiting for a message	Scalasca

$$\#divides > 0 \ \&\& \frac{PM_STALL_FPU}{PM_RUN_CYC} > t \ \&\& vectorized = 0$$

Here, $\#divides$ is the number of divide operations in the loop (collected by some static analysis module or even keywords from the UNIX *grep* utility), while PM_STALL_FPU and PM_RUN_CYC are two metrics collected by HPM that measure the number of cycles spent on stalls due to floating point unit and the total number of cycles, respectively. In this rule t is a constant threshold.

3.2 Metrics from the Compiler

To be able to accurately pinpoint a performance problem, static analysis is oftentimes necessary to understand the structure of the program. It is desirable to bring compiler analysis into the bottleneck discovery. Most compilers are usually not concerned with providing services to external tools, and the complexity of the compiler daunts attempts from outside to utilize its analysis other than compiling a program. Under the DARPA HPCS initiative, currently we are working with the IBM compiler group to expose standard compiler analysis to the tools and users.

Performance metrics provided by the compiler such as estimate of number of prefetchable streams, estimate of pipeline stalls, and number of basic blocks are useful in constructing bottleneck rules. More importantly, since a large class of performance problems are related to optimizations that are not carried out by the compiler (although in theory it is capable of), it is immensely helpful to get a report from the compiler on optimizations performed on the code. Without knowing what optimizations have been applied to the hotspots, it is virtually impossible to provide meaningful diagnostics, especially for problems in the computation domain, as the target binary code becomes a black box to us. Take the *unroll* analysis in our study as an example. Loop *unroll and jam* is a well studied compiler technique. Indeed many compilers claim to have competent unrolling transformations. In practice, however, we found from performance engineers that in many cases even industrial strength compilers do not do the best job in unrolling outer loops. The reasons could be that either only in very high optimization level is outer-loop unrolling triggered (which can be a problem for programs requiring strict semantics), or the compiler decides that other forms of transformation are appropriate which preclude unrolling. As we conduct postmortem analysis of an application and direct tuning efforts to a few code regions, we can afford more detailed analysis for better estimating the costs and benefits of an unrolling vector. Experimental results show that our *unroll* analysis produces faster code than some industrial-strength compiler [4]. In this case, performance bottleneck is the existence of discrepancy between parameters estimated by our module and those proposed by the compiler. A compiler can also report reasons that a potential performance-boosting optimization is not done. Such information provides insight that help further tune an application.

For bottleneck discovery, we utilize analysis results from the compiler are stored in an XML database. Using this database, metrics and transformations on a certain code region can be retrieved.

4 Solution Composition and Implementation

Our framework attempts to propose measures (which we call solutions) to mitigate the performance bottlenecks. Candidate solutions mined from expert knowledge are stored in the solution database. Solutions are in generic forms, and need to be instantiated. For example, a generic solution for the bottleneck where excessive time is spent on blocking MPI calls is to overlap computation with communication, while whether and how the overlap can be done are application dependent. Instantiating a solution involves the following actions. First legality check is necessary to preserve data dependency. Second, the parameter values are computed. In overlapping communication and computation for MPI programs, non-blocking calls and their locations are the parameters. Next, performance improvement is estimated through modeling or running the application patched with the solution. Lastly, code modifications and environment settings are determined.

Parameter values largely determine the effectiveness of the solutions. One important aspect of solution determination is to find the optimal parameter values. Such analysis for CPU related problems is similar to that done by an optimizing compiler. As time constraint on the tuning process of our framework is in general not as stringent as that on that compiler, oftentimes we can search for the optimal parameter values with the help of a performance estimation engine. For example, for loop unroll as a solution, we generate pseudo instructions for different unroll vectors and schedule the instructions to run on an universal machine model [4]. Performance estimation engine counts the number of cycles spent on the normalized loop body, and the search engine chooses the parameter with the best execution cycles for the optimal solution.

The effectiveness of solution discovery is closely related to the accuracy of bottleneck detection. The more detailed a bottleneck description is, the easier for the system to propose solutions. Consider the following scenario. Suppose the framework detects a bottleneck pattern where excessive pipeline stalls occurred in a code region. It is hard to propose any meaningful solution without further information as possible causes are numerous. If it is also detected that the majority of the stalls are due to data cache misses, the hint is to improve data locality. If static analysis reveals that irregular accesses occurred to specific arrays, solution discovery can focus on those arrays. As our framework is involved with every aspect of the tuning process, the quality of solution can be improved by accurate bottleneck discovery.

We implement three kinds of solutions: standard transformations through compilers, modifications to the source code, and suggestions. Compiler support obviates the need to modify the source code for standard transformations. Our framework focuses on searching for better parameter values, and delegates the actual transformation to the compiler. We work with compiler researchers to develop two different facilities for implementing solutions. One is through compiler directives, and the other is through the polyhedral scripting framework [1].

Directives serve as suggestions to the compiler, and the polyhedral framework provides a flexible interface for the tool to implement its own desired optimization composed from standard transformations. Solution in the form of source code modification is necessary when there is no compiler support for the transformations, for example, optimizations to MPI communications. In section 5 we present examples that employ compiler directives and source code modifications to implement solutions.

5 Case Study

Currently our framework contains many built-in metrics and rules for bottleneck detection. Most notably all metrics collected by hardware event counters are present, together with many metrics collected by static analysis and compiler analysis. On the solution side, the framework is able to automatically tune for several performance problems. Here we give an example of using our framework to analyze and tune an application. Note that expanding the framework to automatically tune the two applications provide utilities that are reusable in future tunings.

The application we consider is Lattice Boltzmann Magneto-Hydrodynamics code (LBMHD) [9]. The Lattice Boltzmann method is a mesoscopic description of the transport properties of physical systems using linearized Boltzmann equations.

Hotspot detection shows that two functions, *stream* and *collision*, take most of the execution time. And it is clear that there are many pipeline stalls, and resources in the system are not fully utilized. Tuning experts point to a performance problem that is due to two different access orderings on the same multi-dimension array in *stream* and *collision*.

Figure 11 shows the loops of interest in subroutine *collision*. For multi-dimensional arrays f , g , feq , and geq , the access order incurred by the j , i , k iteration order does not match with their storage order, and creates massive cache misses (consider the k dimension, for example). There are two ways to match the array access order and the storage order. The first is to change the access order by loop-interchange. In *collision*, however, the loops are not perfectly nested. It is impossible to implement loop interchange without violating the dependency constraints. The remaining option is to change the storage order to match the access order by re-laying out the array. Changing the storage order does not affect the correctness of the program, and is a viable solution. Of course, this optimization affects the performance of all accesses to the affected arrays, a fact that needs consideration for solution implementation.

For arrays f and feq , as the control variables are in the k , i , j order counting from the inner most, the new layout is to store the array such that the k dimension is stored first, followed by the i dimension, then the j dimension. In other words, the new storage order is (3,1,2), the 3rd dimension first, then the 1st dimension, followed by the 2nd dimension. For arrays g and geq , all accesses have the forth subscript as constant. The new storage order should store the 4th dimension first. For the k , i , j control variable access order, the storage order is

```

do j = jsta, jend
  do i = ista, iend
    ...
    do k = 1, 4
      vt1 = vt1 + c(k,1)*f(i,j,k) + c(k+4,1)*f(i,j,k+4)
      vt2 = vt2 + c(k,2)*f(i,j,k) + c(k+4,2)*f(i,j,k+4)
      Bt1 = Bt1 + g(i,j,k,1) + g(i,j,k+4,1)
      Bt2 = Bt2 + g(i,j,k,2) + g(i,j,k+4,2)
    enddo
    ...
    do k = 1, 8
      ...
      feq(i,j,k)=vfac*f(i,j,k)+vtauinv*(temp1+trho*.25*vdotc+ &
          .5*(trho*vdotc**2- Bdotc**2))
      geq(i,j,k,1)= Bfac*g(i,j,k,1)+ Btauinv*.125*(theta*Bt1+ &
          2.0*Bt1*vdotc- 2.0*vt1*Bdotc)
      ...
    enddo
    ...
  enddo
  ...
enddo

```

Fig. 1. Code excerpt of collision

similar to that of f , and feq . The final storage order is $(4,3,1,2)$, that is, the 4^{th} dimension first, the 3^{rd} dimension second, followed by the 1^{st} and 2^{nd} dimensions. To implement the new storage order, we resort to compiler directives. For example, on IBM platforms, the *!IBM SUBSCRIPTORDER* directive that accepts a new storage order. For LBMHD, four arrays have their storage orders changed through the following directive.

!IBM SUBSCRIPTORDER($f(3,1,2)$, $feq(3,1,2)$, $g(4,3,1,2)$, $geq(4,3,1,2)$)

Changing the storage order of the arrays may create undesirable side effects for other program constructs. In LBMHD, arrays f , feq , g , and geq are shared by several routines. In our earlier analysis, we found proper storage orders for these arrays in *collision*. However, these orders introduce new problems for *stream* (Figure 2).

The access order in *stream* to arrays g and geq matches exactly with the original storage order. Yet to improve the cache performance for *collision*, the storage orders are changed. To remedy, we can try changing the access order of *stream* to match the new storage order. Changing access order is constraint by data dependency. Fortunately for *stream*, loop-interchange can be applied as follows. The whole nested loop is distributed into two perfectly nested loops. Each in turn is interchanged. Then the two perfectly nested loops are fused together. As a result, the outer most loop (do $k = 1, 2$) is moved to the inner most, the inner most loops (do $i = ista, iend$) to the middle, and the loop (do $j = jsta, jend$) is moved to the outer most.

```

do k = 1, 2
  do j = jsta, jend
    do i = ista, iend
      g(i,j,1,k) = geq(i-1,j,1,k)
      ...
    enddo
  enddo
  do j = jsta, jend
    do i = ista, iend
      g(i,j,2,k) = w1*geq(i,j,2,k) + w2*geq(i-1,j-1,2,k)
      + w3*geq(i-2,j-2,2,k)
      g(i,j,4,k) = w1*geq(i,j,4,k) + w2*geq(i+1,j-1,4,k)
      + w3*geq(i+1,j-2,4,k)
      ...
    enddo
  enddo
enddo

```

Fig. 2. Code excerpt from *stream*

Using our bottleneck description language, the following rule is inserted to the bottleneck database for detecting the mismatch between iteration order and storage order.

$$\begin{aligned}
& \text{STALL_LSU/PM_CYC} > \alpha \text{ and } \text{STRIDE1_RATE} \leq \beta \\
& \text{and } \text{REGULAR_RATE}(n) > \text{STRIDE1_RATE} + \gamma
\end{aligned}$$

STALL_LSU and PM_CYC are metrics that measure the number of cycles spent on LSU stalls and the total number of cycles, respectively. STRIDE1_RATE estimates the number of memory accesses that are stride-1. REGULAR_RATE estimates the number accesses that have regular stride. What this rule says is that if there is a significant number of cycles spent on LSU unit, and there are more n -stride accesses than stride-1 access, there is potentially a bottleneck that may be removed by array transpose and related optimizations. In the rule, α , β , and γ are constants used as thresholds. Different threshold values signify the different levels of seriousness of the performance problem. The threshold values may also be tied to the performance gain a user expects from removing the bottleneck. Static analysis and runtime hardware events collection modules are responsible for the collection of these metrics.

There is only one solution associated with this bottleneck, as the bottleneck definition narrows the range of possible solutions. Discovering solution parameters and implementing the solution are still manual at this stage. Yet after the development of these modules, utilities such as finding the array access order and the storage order become available to further users, and can simplify their development of similar modules.

Once the bottleneck rule and solution discussed above are introduced, our framework can automate the performance tuning for LBMHD. Similar

performance problems can be detected and mitigated for other applications. Due to limited space, we refer interested readers to IBM *alphaworks* [8] for examples and instructions of using the framework.

In our experiment, the *gtranspose* solution achieved about 20% improvement in execution time with a grid size 2048×2048 and 50 iterations on a P575+ (1.9 GHz Power5+, 16 CPUs. Memory: 64GB, DDR2) on one processor. Note that without loop-interchange for *stream*, the transformation actually degrades the overall performance by over 5% even though the performance of *collision* is improved. Loop-interchange mitigates the adverse impact of the new storage order on *stream*. There is still performance degradation of *stream* after Loop-interchange. The degradation is due to the new memory access pattern. After Loop-interchange, the k -loop becomes the inner-most loop. Although the memory accesses to g and geq are consecutive, the corresponding array dimensions are of a small range (k goes from 1 to 2). And the next dimension introduces strided access. In the original code, both the i -loop and j -loop have sequential accesses to g and geq spanning a region of size $(iend - ista) \times (jend - jsta)$.

6 Conclusion and Future Work

In this paper we presented our study of unifying performance tools, compiler, and expert knowledge for high productivity performance tuning. Our framework facilitates a holistic approach that detects bottlenecks and proposes solutions. Performance data collected by existing performance tools can be used as metrics. The analysis of multiple tools can be correlated and combined through bottleneck rules. Compiler analysis and optimization play a critical role in our framework. We demonstrated the advantages of performance optimization through our framework through tuning two applications. The tuning process is highly automated, and the performance improvement is significant in both cases.

As we observed, the effectiveness of the framework depends on the number and quality of bottleneck rules and solutions in our database. In future work, we plan to populate the database with more rules and solutions. We also expect to improve the services and utilities the framework provides for expansion and customization.

References

1. Bastoul, C.: Code generation in the polyhedral model is easier than you think. In: Proc. 13th international conference on parallel architecture and compilation techniques, Antibes Juan-les-Pins, France, September 2004, pp. 7–16 (2004)
2. Bhatele, A., Cong, G.: A selective profiling tool: towards automatic performance tuning. In: Proc. 3rd Workshop on System Management Techniques, Processes and Services (SMTPS 2007), Long beach, California (March 2007)
3. Chen, W., Bringmann, R., Mahlke, S., et al.: Using profile information to assist advanced compiler optimization and scheduling. In: Banerjee, U., Gelernter, D., Nicolau, A., Padua, D.A. (eds.) LCPC 1992. LNCS, vol. 757, pp. 31–48. Springer, Heidelberg (1993)

4. Cong, G., Seelam, S., et al.: Towards next-generation performance optimization tools: A case study. In: Proc. 1st Workshop on Tools Infrastructures and Methodologies for the Evaluation of Research Systems, Austin, TX (March 2007)
5. DeRose, L., Ekanadham, K., Hollingsworth, J.K., Sbaraglia, S.: Sigma: a simulator infrastructure to guide memory analysis. In: Proceedings of the 2002 ACM/IEEE conference on Supercomputing, pp. 1–13 (2002)
6. Geimer, M., Wolf, F., Wylie, B.J.N., Abraham, E., Becker, D., Mohr, B.: The SCALASCA performance toolset architecture. In: Proc. Int'l Workshop on Scalable Tools for High-End Computing (STHEC), Kos, Greece (2008)
7. High productivity computer systems (2005), <http://highproductivity.org>
8. High productivity computing systems toolkit. IBM alphaworks, <http://www.alphaworks.ibm.com/tech/hpcst>
9. MacNab, A., Vahala, G., Pavlo, P., Vahala, L., Soe, M.: Lattice Boltzmann Model for Dissipative Incompressible MHD. In: 28th EPS Conference on Contr. Fusion and Plasma Phys., vol. 25A, pp. 853–856 (2001)
10. Malony, A.D., Shende, S., Bell, R., Li, K., Li, L., Trebon, N.: Advances in the tau performance system, pp. 129–144 (2004)
11. Miller, B.P., Callaghan, M.D., Cargille, J.M., Hollingsworth, J.K., Irvin, R.B., Karavanic, K.L., Kunchithapadam, K., Newhall, T.: The Paradyne Parallel Performance Measurement Tool. IEEE Computer 28, 37–46 (1995)
12. Mohr, B., Wolf, F.: KOJAK – A tool set for automatic performance analysis of parallel programs. In: Kosch, H., Böszörményi, L., Hellwagner, H. (eds.) Euro-Par 2003. LNCS, vol. 2790, pp. 1301–1304. Springer, Heidelberg (2003)
13. Pillet, V., Labarta, J., Cortes, T., Girona, S.: PARAVER: A tool to visualise and analyze parallel code. In: Proc of WoTUG-18: Transputer and occam Developments, vol. 44, pp. 17–31. IOS Press, Amsterdam (1995)
14. Vuduc, R., Demmel, J., Yelick, K.: OSKI: A library of automatically tuned sparse matrix kernels. In: Proceedings of SciDAC 2005, Journal of Physics: Conference Series (2005)
15. Wen, H., Sbaraglia, S., Seelam, S., Chung, I., Cong, G., Klepacki, D.: A productivity centered tools framework for application performance tuning. In: QEST 2007: Proc. of the Fourth International Conference on the Quantitative Evaluation of Systems (QEST 2007), Washington, DC, USA, 2007, pp. 273–274. IEEE Computer Society, Los Alamitos (2007)
16. Whaley, R., Dongarra, J.: Automatically tuned linear algebra software (ATLAS). In: Proc. Supercomputing 1998, Orlando, FL (November 1998), www.netlib.org/utk/people/JackDongarra/PAPERS/atlas-sc98.ps

Pattern Matching and I/O Replay for POSIX I/O in Parallel Programs

Michael Kluge, Andreas Knüpfer, Matthias Müller, and Wolfgang E. Nagel

Technische Universität Dresden, Dresden, Germany

{michael.kluge, andreas.knuepfer}@tu-dresden.de,

{matthias.mueller, wolfgang.nagel}@tu-dresden.de

Abstract. This paper describes an approach to track, compress and replay I/O calls from a parallel application. We will introduce our method to capture calls to POSIX I/O functions and to create I/O dependency graphs for parallel program runs. The resulting graphs will be compressed by a pattern matching algorithm and used as an input for a flexible I/O benchmark. The first parts of the paper cover the information gathering and compression in the area of POSIX I/O while the later parts are dedicated to the I/O benchmark and to some results from experiments.

1 Introduction

Parallel file systems are an essential part of large HPC systems. Using such an infrastructure efficiently is a key factor for a high utilization. Yet, the requirements and usage patterns concerning the I/O infrastructure vary from application to application. Planning the exact demands of future file systems, e.g. during a procurement of a new HPC system, depends on the knowledge about the I/O behavior of the applications that will run on those machines. Generally each file system design is measured on two criteria: bandwidth and I/O operations per second. Both values are presented often but are not representative for most applications because they neither generate a lot of I/O requests nor use very large block sizes. Beside artificial benchmarks the ability to exactly simulate the I/O behavior of an application is very valuable.

The outline of this paper is defined by the path towards a better understanding of I/O in an application and complete I/O replay. At first we will explain how to collect all necessary information about all I/O calls from an application. Furthermore we show how the pattern matching algorithm helps us to reduce this data to a reasonable amount. In the second part we will go into the details of the design and the implementation of our file system benchmark 'DIOS'. The last part of the paper covers experimental results and the outlook.

2 Methods and Data Structure

In order to replay the I/O activities of an application we need to record those activities first. For this step we employ a modified version of the OTF library [2] together with VampirTrace [20].

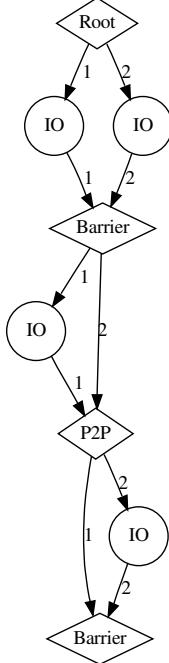


Fig. 1. example for two processes

an example of two processes both executing POSIX I/O in a first phase in parallel but individually in the second and third phase.

The next section covers the reduction of the amount of synchronization events within the data structure by an heuristic approach.

3 Reduction of Synchronization Events

The aim of the analysis process described in this paper is to extract all I/O activities from an application trace. From there we want to collect all information needed to replay this I/O operations without the original program. The information needed for the purpose of preserving the logical order of the I/O events is: Was there any (relevant) synchronization between consecutive I/O events for a process? After the construction of the dependency graph a lot of applications showed the same kind of patterns if seen from a very high level: Patterns of

Logging the arguments of function calls is not supported natively by OTF or VampirTrace. We use OTF comments and an enhanced version of the VampirTrace I/O tracing facilities to write one comment for each argument. After the execution of the instrumented version of the program the trace is read (post mortem) and all synchronization and I/O events are put into a data structure that preserves the logical order of the events. Among parallel processes the logical order of I/O events needs to be conserved because they may not be independent.

One obvious example supporting this constraint is where one process writes a file, all processes synchronize and use this file afterwards. The data structure is stored as a directed acyclic graph with two types of nodes, I/O nodes and synchronization nodes. The edges that connect nodes together describe the logical order of the events. I/O nodes contain one or more I/O events between two synchronization points. Synchronization nodes always have one edge leading into the node and one edge pointing to next node for each process that is part of this event. Fig. II shows

synchronization events and intermixed a few I/O nodes. So it is very likely that there are redundant synchronizations events within the data structure that can be removed without violating the integrity of the I/O patterns. From a given program structure there can be multiple possibilities to remove one or more synchronization events. For our experiments we have chosen an heuristic approach: All synchronization nodes that have only one parent node (with multiple edges between the two nodes) are being removed from the graph. With a more formal approach it would be possible to reduce the number of synchronization events even more, yet with an increased computational effort. After reducing the number of synchronization nodes in the graph the next section describes our approach to compress the graph on a per-process base.

4 Pattern Matching for Parallel I/O Patterns

The pattern matching algorithm used in this paper is applied in a post processing step. All tokens from I/O and synchronization events are collected in a compression queue where each process has its own queue.

An element in the queue is either an I/O operation or an synchronization event. Both types are being represented as hash values. Function calls with the same list of arguments result in the same hash value, collisions are not possible due to the hashing function. Every time one element is added, the algorithm tries to match the tail of the queue against all known patterns that have been used up to that point in time. As long as the tail is being replaced with a pattern from the known pattern list, this operation is repeated. Afterwards we apply the compression algorithm as described in [17]. The whole step is repeated until no further compression is possible. The modified algorithm outline is depicted in Fig. 2.

Unlike [17] we do not perform compression across processes. Therefore the result is a compressed representation of I/O activity on a per-process base that serves as input for the I/O benchmark. An alternative way for compression

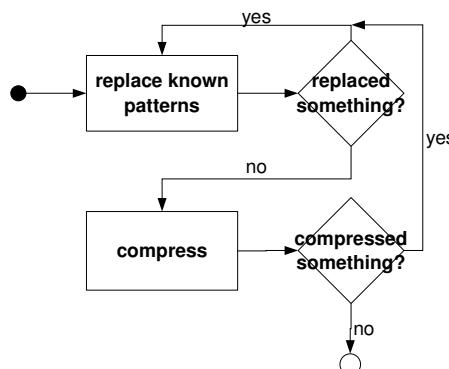


Fig. 2. structure of the compression algorithm

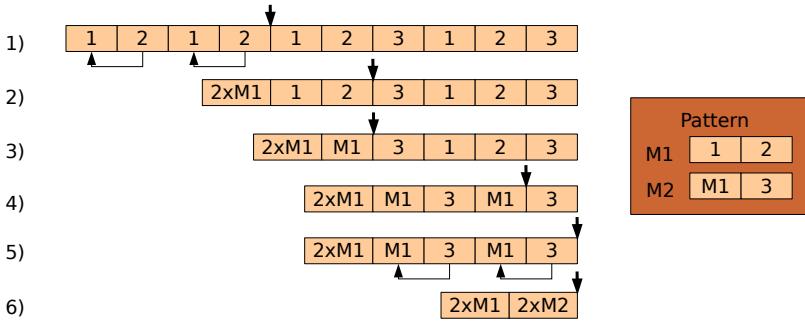


Fig. 3. example run of compression algorithm for the sequence {1,2,1,2,1,2,3,1,2,3}

could be searching for the longest common subsequence [4], but with higher computational effort.

Fig. 3 shows an example of the compression scheme. The arrow above the line shows the portion of the input sequence that has been read by the compression algorithm at each individual point. At step 1) the compression scheme recognizes a repetition and will create pattern M1 and replace the four elements with 'two times M1'. From this point in time the subsequence {1,2} is always replaced with 'M1' (steps 4 and 5). Step 6 shows how a new pattern that includes known patterns as is created.

5 I/O Replay with DIOS

The DIOS benchmark has been designed to execute any given (parallel) I/O load. To provide this kind of flexibility, the C++ software is divided into three essential parts (see also Fig. 4). The central part of the code executes POSIX and MPI I/O calls that have been specified in an input file. The parsing and preprocessing of the input file is another part of the software. The last part is a flexible backend that has a 'begin' and an 'end' handler for each implemented I/O call. The benchmark is able to run in parallel and each process can have different I/O requests to execute. Optionally, it is also capable of checking the content read from a file for errors. This is slower than running without checking the file content but helps investigating data corruption problems.

The backend gets notice of the beginning and the end of all I/O operations for all tasks as well as additional information specific to each executed I/O request (like the number of bytes written with a call to `write()`). This way the backend can generate traces or calculate statistics of all I/O activities. The DIOS benchmark is being shipped with different backends, the following are currently available:

EmptyLogger prints nothing, useful if only the imitation of application I/O is needed, e.g. as background load or for data corruption tests

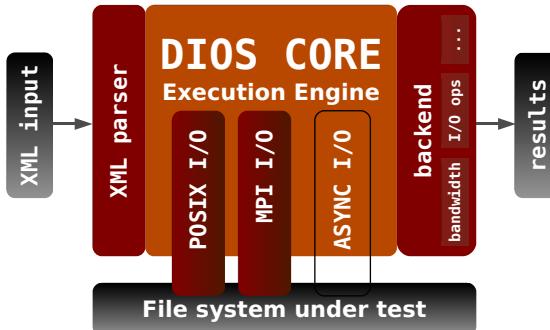


Fig. 4. DIOS software structure

Read-/WriteBandwidth prints the total read or write bandwidth at the end of the run, defined as total bytes transferred divided by the total runtime of the benchmark

ClientRead-/WriteBandwidth like Read-/WriteBandwidth but does only use the time spent within I/O operations instead of the full runtime.

ChunkBandwidth prints the average bandwidth separately for each I/O request size

CreateUnlink prints statistics how long it took (min, max, mean, average) to create files and to unlink files

It is possible to extend this list easily using a C++ interface. Each backend class has two additional functions that are called at program startup and after all I/O requests have been executed. The first can be used to initialize the backend. The other is commonly used to evaluate the collected data and to print the results of the evaluation. The benchmark framework also includes a common timer function `clockticks()`.

Within tests we have seen no difference between the bandwidth reported by micro-benchmarks and DIOS when performing the same I/O operations.

6 Input File Description

As format for the input file we have chosen XML. XML is easy to handle and can be checked for integrity automatically during the parsing process. The file is being parsed using libxml2 [21] and contains a description of all operations that shall be performed by the benchmark.

Within the input file we differentiate between three layers of information. The first layer describes what I/O operations every task has to execute. The second level describes a series of event sets. Event sets are the third information layer and describe a specific pattern to execute that originated from the compression step, see Sect. 4. The additional 'series' layer has been introduced to be able to repeat multiple patterns, each described within its own event set. The actual

```

iotest      = (task|mpi_datatype)*

task        = (series,mpi_series)*

mpi_series = (mpi_fopen|mpi_fclose|mpi_fdelete|mpi_set_view|
              mpi_read|mpi_write|mpi_wait|sleep|mpi_series)+

series      = (fopen|open|fclose|close|buf_read|buf_read_random|
              buf_write|buf_seek|read|write|seek|sync|sleep|
              poolopen|poolrandom|unlink|pread|pwrite|series)+
```

Fig. 5. Simplified EBNF rules for the input file (subset needed for I/O replay)

I/O operations are encoded as leaves in this XML tree within the event set. A simplified overview about the supported operations as well as the internal structure is given in Fig. 5 as EBNF rules.

Several parameters within the XML file can not just be numbers, they can be expressions with variables inside. There are two predefined variables 'SIZE' and 'RANK' where the first keeps the total number of tasks running and the second has an unique ID per running task in the range $[0 \dots SIZE - 1]$. For example, to grow a file linearly with the number of tasks that write a portion of this file it is possible to say 'task N seeks to position $N \cdot X$ in the file and writes X bytes', where X is a fixed amount of bytes that each task is writing to the file. Other possibilities include for example the definition of an XML file that would write always the same file size, but with different numbers of tasks.

New variables can be defined by setting an environment variable with the same name. This way it is possible to create generalized and reusable XML files which is beyond the scope of the input generated by the compression scheme (Sect. 4) but allows to generalize its results very easily.

One further concept implemented in the benchmark are file pools. File pools have been designed to have a collection of file names available on a variable directory structure without the need to define each file by hand within the XML file. For tests that create one million files and unlink those files afterwards one can just create a file pool of that size and open each file in that pool. File pools are able to create all the files in the file pool upon initialization. The file sizes for that initialization process can be chosen between a minimum and a maximum. The individual file size for each file will be a random number between this minimum and this maximum. It is also possible to step over this initialization. This is needed for create/unlink tests where the files should not be created or touched in advance. File pools are able to spread the associated files among subdirectories.

For MPI-I/O we have implemented the XML leave elements as described in Sect. 5. According to this we access all calls to MPI2 read functions using one XML tag `<mpi-read>` with the attributes `pos`, `sync`, `coord`. This way we can easily switch between different ways to access data with different MPI-I/O routines. Another option would have been to implement all MPI-I/O functions and provide those as tags within the XML structure. By combining all read

functions into one tag it is much easier to evaluate the various ways of reading data from a file with MPI-I/O. Further on, we implemented an interface to describe derived MPI data types in the XML file and to use those to test their effects on MPI I/O calls.

7 Related Work

This section covers a comparison with other I/O tracing tools as well as related work in the areas of I/O replay and benchmarks.

Taking [15] as a reference, our approach as a whole fits into the 'script based I/O benchmark' category as well as the similar approach in the //Trace tool [16]. //Trace is collects and replays I/O events. It supports a discovery of all dependencies between I/O requests at high costs (N instrumented program runs with N processors) and does not support a compression of the resulting trace files. The pattern matching used in our paper allows researchers to gain additional information about the I/O part of an application, like insight into I/O patterns.

An other approach to generate I/O traces is tracefs [22]. The approach to trace I/O from the VFS layer in the Linux Kernel is an elegant way to catch memory mapped I/O as well. Nevertheless the information generated here still have to be merged into the application trace in order to learn more about the application. Integrating traces of memory mapped I/O into application I/O traces would be very beneficial. replayfs is an extend to tracefs and allows to replay the recorded VFS trace from the VFS layer and not from user space. This allows to examine different file systems and storage hardware in more detail but is not suitable for the examination of applications.

Compared to an approach as presented in [1], we do not need manual source code instrumentation (but is available with VampirTrace as well) and try to find patterns automatically that could represent loop structures in the source code. However, our tool does not do performance modeling (scalability analysis etc.) yet.

The most commonly used benchmarks for evaluating parallel file systems seem to be `b_eff.io` [19] and `IOR` [14]. Results for both benchmarks are commonly shown when parallel file systems and large HPC systems in general are being compared. `b_eff.io` tests a lot of different access patterns as well as 'bad' I/O with different numbers of processors and buffer size. Those parameters can be adjusted to achieve best performance on different HPC systems.

`IOR` has a lot of command line parameters and can also be driven by a script file that allows to combine a number of command line options to one (or multiple) measurement steps. One thing that singles out `IOR` above other benchmarks is the ability to perform online outlier checks. Tasks that perform slower than other tasks can be identified at runtime.

One benchmark that is able to generate a parallel I/O workload is `IOAgent` [11]. `IOAgent` is able to execute patterns of I/O operations on a file. This I/O operations have different properties in terms of operation type, buffer size and concurrency. The size of I/O operations is fixed for each running task and the execution time of operations with a task can be influenced by statistical distributions.

Other HPC benchmarks that have been designed to evaluate large file systems are PRIOmark [13] or HPIO [6]. Both are more synthetic workloads that have been designed to be able to compare file systems where HPIO is more targeted to noncontiguous distributed data.

Although more aiming at desktop systems, other file system benchmarks that could be used in an HPC environment are IOzone [18] or bonnie++ [8].

In contrast to those more general benchmarks more application specific benchmarks exist. Flash I/O [23], NPB I/O (BTIO) [3] or mpi-tile-io [9] have been derived or designed to represent only the I/O portion of a more complex application. Due to this they are limited to investigate one problem.

Provided appropriate XML input files, the DIOS benchmark is able to mimic all of the other benchmarks behavior.

8 Examples

At this point we present some real world examples where we have applied the tools presented. The first example is a well known bioinformatic applications while the second one is a CFD code.

The bioinformatic application we were looking at is mpiBLAST [10]. BLAST is being used for genomic sequence search. mpiBLAST is an open-source implementation of NCBI BLAST that uses MPI tasks to search a segmented database in parallel. The database is being split into independent parts (files) in a serial post-processing step before the actual parallel run of mpiBLAST. This step is done by `mpiformatdb`. It reads a big part of the input database at once in the beginning and performs some processing using a temporary file and writes the separated database files to disk. The excerpt from the I/O pattern (see Fig. 6) shows the initial reading of the database and some patterns where the temporary file is filled and more input from the database is read into the application.

```

1 ...
2     1:     fopen64( drosoph.nt, r);
3 1534943:     fgets( 0x607fffffe796ff0, 10000, 3);
4     1:     fopen64( /tmp/reorder0zJDIo, w);
5     1:     fseek( 3, 88598402, 0);
6 34:[
7     1:         fread( 0x607fffffe797040, 1, 10000, 3);
8     1:         fwrite( 0x607fffffe797040, 1, 10000, 5);
9     ]
10    1:         fread( 0x607fffffe797040, 1, 8234, 3);
11    1:         fwrite( 0x607fffffe797040, 1, 8234, 5);
12    1:         fread( 0x607fffffe797040, 1, 0, 3);
13    1:         fseek( 3, 96528234, 0);
14 34:[
15     1:         fread( 0x607fffffe797040, 1, 10000, 3);
16     1:         fwrite( 0x607fffffe797040, 1, 10000, 5);
17     ]
18 ...

```

Fig. 6. Excerpt of an I/O pattern: `mpiformatdb`. The [denotes the start of a repeated pattern, the] denotes the end of a pattern and the number before the colon are the number of repetitions for this pattern.

```

1 ...
2 1:      fopen64( TMF_Nz1024.rst, r);
3 1:      read( 15, ptr, 8191);
4 1:      read( 15, ptr, 1432160);
5 3:      read( 15, ptr, 1440000);
6 4:[
7 1:      read( 15, ptr, 1440000);
8 1:      send_sync( 2);
9 ]
10 4:[
11 1:      read( 15, ptr, 1440000);
12 1:      send_sync( 3);
13 ]
14 4:[
15 1:      read( 15, ptr, 1440000);
16 1:      send_sync( 4);
17 ]
18 4:[
19 1:      read( 15, ptr, 1440000);
20 1:      send_sync( 5);
21 ]
22 ...

```

Fig. 7. I/O pattern in Semtex for process 0 for the serialized distribution of the input data

The second example Semtex [5], [12] is a CFD application that is using millions of CPU hours at the TU Dresden. It is a family of spectral element simulation codes. The spectral element method is a high-order finite element technique that combines the geometric flexibility of finite elements with the high accuracy of spectral methods. From the I/O point of view, Semtex has two interesting phases. The first one is the startup phase where the input data (or the restart file from a checkpoint) is being read. The first look at the pattern obtained from a program run with 256 processors reveals a serialized distribution of the input file which is read completely by process 0 (see Fig. 7). (The corresponding part for the other 255 processes is always just `recv_sync(1)`).

The other interesting phase is the checkpointing phase where we have seen another serialization as only process 0 is writing the checkpointing file. The pattern matching does not yet use loop variables. Applying this idea would result in shorter representation of patterns like in Fig. 7 which could also be presented as in Fig. 8.

```

1 ...
2 1:      fopen64( TMF_Nz1024.rst, r);
3 1:      read( 15, ptr, 8191);
4 1:      read( 15, ptr, 1432160);
5 3:      read( 15, ptr, 1440000);
6 4:[
7 1:      read( 15, ptr, 1440000);
8 1:      send_sync( x);
9 ]
10 ]{x=2...256}
11 ...
12 ...

```

Fig. 8. Shortened I/O pattern for Fig. 7

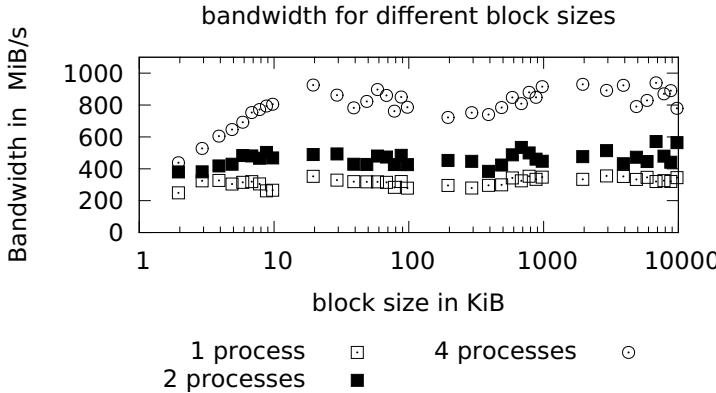


Fig. 9. parallel CXFS file system performance with different block sizes and process count

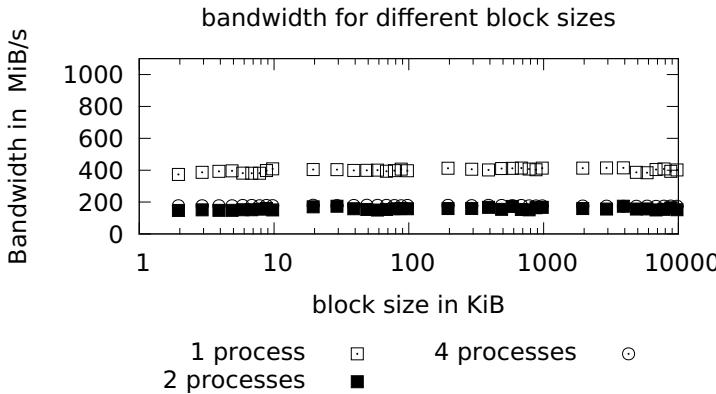


Fig. 10. scratch XFS file system performance with different block sizes and process count

Both phases use chunk sizes of 1.4 MB which is rather large. The bandwidth from the local file system on our test machine (SGI Altix 4700, 2048 Cores, CXFS with 640 spinning disks and 10 GB/s peak I/O bandwidth, local scratch with XFS and 8 spinning disks and 600 MB/s peak I/O bandwidth) is about 160-170 MB/s on CXFS and 310-320 MB/S in XFS.

An investigation with DIOS showed that for this kind of I/O operations¹ the CXFS bandwidth would only grow with more parallel processes and not with larger chunk sizes (see Fig. 9).

The local scratch file system on the other hand has problems handling parallel threads (see Fig. 10). The layout of the underlying md device is not known to the

¹ C++ buffered streams.

XFS file system². Therefore XFS can not exploit its full capabilities [7]. Optimizing the applications I/O behavior would therefore lead to different strategies for both file systems.

9 Conclusion

This paper presented a pattern detection approach for I/O operations and the usefulness of those patterns for detecting serialization points in applications. Beyond this, we have also introduced our I/O benchmark that is able to do a replay of this I/O activities in order to find the best file system for a given I/O pattern or to estimate the I/O performance of an application on a new HPC system. The revealed I/O patterns have immediately given us insight into the I/O part of the investigated applications. We want to thank Frank Müller from North Carolina State University for discussions at the startup phase of this research.

References

1. Alam, S.R., Vetter, J.S.: A framework to develop symbolic performance models of parallel applications. In: Parallel and Distributed Processing Symposium, IPDPS 2006 (April 2006)
2. Knüpfer, A., Brendel, R., Brunst, H., Mix, H., Nagel, W.E.: Introducing the Open Trace Format (OTF). In: Alexandrov, V.N., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2006. LNCS, vol. 3992, pp. 526–533. Springer, Heidelberg (2006)
3. Bailey, D., Barszcz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R., Fineberg, S., Frederickson, P., Lasinski, T., Schreiber, R., Simon, H., Venkatakrishnan, V., Weeratunga, S.: NAS Parallel Benchmarks (2007), <http://www.nas.nasa.gov/News/Techreports/1994/PDF/RNR-94-007.pdf>
4. Bergrøth, L., Hakonen, H., Raita, T.: A survey of longest common subsequence algorithms. In: Proceedings of Seventh International Symposium on String Processing and Information Retrieval, 2000. SPIRE 2000, pp. 39–48 (2000)
5. Blackburn, H.M., Sherwin, S.J.: Formulation of a Galerkin spectral element-Fourier method for three-dimensional incompressible flows in cylindrical geometries. *J. Comp. Phys.* 197, 759–778 (2004)
6. Ching, A., Choudhary, A., Liao, W., Ward, L., Pundit, N.: Evaluating I/O Characteristics and Methods for Storing Structured Scientific Data. In: Proceedings of IPDPS (2006)
7. Chinner, D., Higdon, J.: Exploring High Bandwidth Filesystems on Large Systems. Technical report, Ottawa Linux Symposium, SGI (2006)
8. Cocker, R.: Bonnie++ web page (2001), <http://www.coker.com.au/bonnie++>
9. Parallel I/O Benchmarking Consortium. Parallel I/O Benchmarking Consortium web site (2007), <http://www-unix.mcs.anl.gov/pio-benchmark>
10. Darling, A.E., Carey, L., Feng, W.c.: The design, implementation, and evaluation of mpiblast. In: 4th International Conference on Linux Clusters: The HPC Revolution 2003 (2003)

² Has not been given on the mkfs.xfs command line.

11. Gómez-Villamor, S., Muntés-Mulero, V., Pérez-Casany, M., Tran, J., Rees, S., Larriba-Pey, J.-L.: IOAgent: A parallel I/O workload generator. In: Nagel, W.E., Walter, W.V., Lehner, W. (eds.) Euro-Par 2006. LNCS, vol. 4128, pp. 3–14. Springer, Heidelberg (2006)
12. Koal, K., Stiller, J., Grundmann, R.: Linear and nonlinear instability in a cylindrical enclosure caused by a rotating magnetic field. *Phys. Fluids* 19, 088107 (2007)
13. Krietemeyer, M., Merz, M.: IPACS-Benchmark - Integrated Performance Analysis of Computer Systems (IPACS). Logos Verlag, Berlin (2006)
14. Loewe, W., Morrone, C.: IOR benchmark (2007),
<http://www.llnl.gov/icc/lc/siop/downloads/download.html>
15. May, J.: Pinola: A script-based i/o benchmark. In: Petascale Data Storage Workshop at SC 2008 (2008)
16. Mesnier, M.P., Wachs, M., Sambasivan, R.R., Lopez, J., Hendricks, J., Ganger, G.R., O'Hallaron, D.R.: Trace: parallel trace replay with approximate causal events. In: FAST 2007: Proceedings of the 5th USENIX conference on File and Storage Technologies, p. 24. USENIX Association (2007)
17. Noeth, M., Mueller, F., Schulz, M., de Supinski, B.R.: Scalable compression and replay of communication traces in massively parallel environments. In: IPDPS, pp. 1–11 (2007)
18. Norcott, W.D., Capps, D.: IOzone benchmark (2006), <http://www.iozone.org>
19. Rabenseifner, R., Koniges, A.E.: Effective File-I/O Bandwidth Benchmark. In: Bode, A., Ludwig, T., Karl, W.C., Wismüller, R. (eds.) Euro-Par 2000. LNCS, vol. 1900, p. 1273. Springer, Heidelberg (2000)
20. ZIH TU Dresden. VampirTrace (2007),
<http://www.tu-dresden.de/zih/vampirtrace>
21. Veillard, D.: The XML C parser and toolkit (2007), <http://www.xmlsoft.org>
22. Wrigth, C.P., Aranya, A., Zadok, E.: Tracefs: A file system to trace them all. In: Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004), pp. 129–143. USENIX Association (2004)
23. Zingale, M.: FLASH I/O benchmark routine – parallel HDF 5 (2001),
http://www.ucolick.org/~zingale/flash_benchmark_io

An Extensible I/O Performance Analysis Framework for Distributed Environments

Benjamin Eckart¹, Xubin He¹, Hong Ong², and Stephen L. Scott²

¹ Tennessee Technological University, Cookeville, TN

² Oak Ridge National Laboratory, Oak Ridge, TN

Abstract. As distributed systems increase in both popularity and scale, it becomes increasingly important to understand as well as to systematically identify performance anomalies and potential opportunities for optimization. However, large scale distributed systems are often complex and non-deterministic due to hardware and software heterogeneity and configurable runtime options that may boost or diminish performance. It is therefore important to be able to disseminate and present the information gleaned from a local system under a common evaluation methodology so that such efforts can be valuable in one environment and provide general guidelines for other environments. Evaluation methodologies can conveniently be encapsulated inside of a common analysis framework that serves as an outer layer upon which appropriate experimental design and relevant workloads (benchmarking and profiling applications) can be supported.

In this paper we present ExPerT, an *Extensible Performance Toolkit*. ExPerT defines a flexible framework from which a set of benchmarking, tracing, and profiling applications can be correlated together in a unified interface. The framework consists primarily of two parts: an extensible module for profiling and benchmarking support, and a unified data discovery tool for information gathering and parsing. We include a case study of disk I/O performance in virtualized distributed environments which demonstrates the flexibility of our framework for selecting benchmark suite, creating experimental design, and performing analysis.

Keywords: I/O profiling, Disk I/O, Distributed Systems, Virtualization.

1 Introduction

Modern computing paradigms, of which distributed computing represents perhaps the most large-scale, tend to be extremely complex due to the vast amount of subsystem interactions. It is no surprise then that it is often very difficult to pinpoint how a change in system configuration can affect the overall performance of the system. Even the concept of performance itself can be subject to scrutiny when considering the complexities of subsystem interactions and the many ways in which performance metrics can be defined. Such varying definitions coupled with the combinatorially large array of potential system configurations

create a very hard problem for users wanting the very best performance out of a system.

A potential solution then is to provide a framework flexible enough to accommodate changing technologies and to be able to integrate advancements in I/O performance analysis in a complementary way. Many complex programs exist to benchmark and profile systems. There are statistical profilers, such as OProfile [1], sysprof [2], qprof [3], source level debugging tools, such as gprof [4], system-level statistics aggregators, such as vmstat [5] and the sysstat suite [6], and various macro, micro and nano-benchmarking tools, such as Iozone [7], lmbench [8], bonnie++ [9], iperf [10], and dbench [11]. All of these tools measure different computing elements. It would be convenient to have a common framework through which results from any test could be aggregated in a consistent and meaningful way. Furthermore, such a framework would be able to avoid “re-inventing the wheel” since it should be able to accommodate new developments in a modular fashion. *In short, our motivation is toward a software framework that would provide the flexibility to support relevant workloads, appropriate experiments, and standard analysis.* As of this writing, we are unaware of any framework currently exists for these sort of tests, but such a tool, if properly designed, could greatly help our understanding of such systems.

Our contributions detailed in this paper include the following: We design a flexible, extensible, and unified framework for I/O characterization in distributed environments. ExPerT facilitates system-wide benchmarking and profiling and provides tools for analysis and presentation, such as automatic graph generation, detailed performance statistics, and other methods of data discovery. The design itself is modular and extensible, being able to incorporate current state of the art benchmarking and profiling tools. ExPerT is designed to be usable in large distributed or high performance environments, with jobs transparently able to be executed in parallel and on different machines using lightweight TCP clients dispersed to each node.

The rest of this paper continues as follows: In Section 2 we discuss the design goals, implementation, and functionality of the framework. In Section 3, we use ExPerT to do preliminary analysis of disk I/O on both the KVM and Xen virtualization platforms. Section 4 introduces related work in the areas of benchmarking and analysis. Lastly, in Section 5 we give our conclusions.

2 I/O Characterization Framework

With goals of modularity and extensibility, we constructed ExPerT to support multiple profilers and benchmarking applications.

Our framework consists primarily of two components: a testing and batch creation tool, (*batch*), and a data discovery tool, (*mine*). A high-level view of these components can be seen in Figure 1. The *batch* module serves as a wrapper around the *test* module. Single tests can then be represented simply as a singleton batch test. The *mine* module parses and interactively enables a user to produce graphs and statistics from the tests.

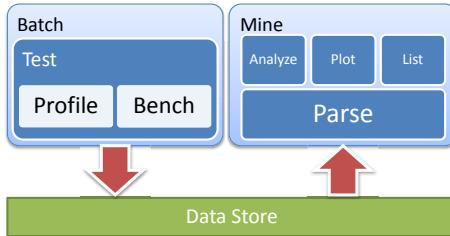


Fig. 1. Two main components comprise the framework: *batch* deals with the production of data. *mine* deals with aspects related to the consumption of the data: data parsing, trend discovery, and graphical plotting tools. Both share data via the lightweight database, *sqlite*.



Fig. 2. The *batch* component: *batch* serves as a wrapper around *test*, which is itself comprised of three components, *prerun*, *run*, and *postrun*. *run* consists of an array of threads with a single master and n slaves. When the master stops, it signals the other threads to stop in unison. The threads comprising *run*, along with the *prerun* and *postrun* sections, can be a mixture of local and remote operations.

A graphical overview of the batch testing tool is shown in Figure 2. The testing tool defines three main functions that occur for each benchmarking or profiling application. First, a sequential list of commands can optionally be executed. This constitutes the *prerun* segment. The *prerun* module can contain any number of local or remote commands that prepare the system for the test. An example for disk I/O testing would be a *dd* command that constructs a file for benchmarking disk read throughput. The *run* segment starts an array of threads simultaneously, with each given local or remote commands for profiling and benchmarking. The threads follow a strict hierarchy: all threads begin at the same time, but all but one are designated to be “slave” threads. No threads halt execution until the single “master” thread halts. Since the master thread is designed to coordinate all the other threads synchronously, it suitably should be some terminal benchmark or application that is set to halt after a predetermined amount of time. The slave threads typically include profilers and tracers. Thus, we can “benchmark” benchmarks as a way of aggregating statistics from different tools. For example, we can run a single disk I/O test in *Iozone* and simultaneously watch the context-switching effects with *vmstat*. Finally, we conclude the test with another batch of sequentially executed commands. The implementation of this synchronization functionality utilizes a lightweight TCP client dispersed to each node.

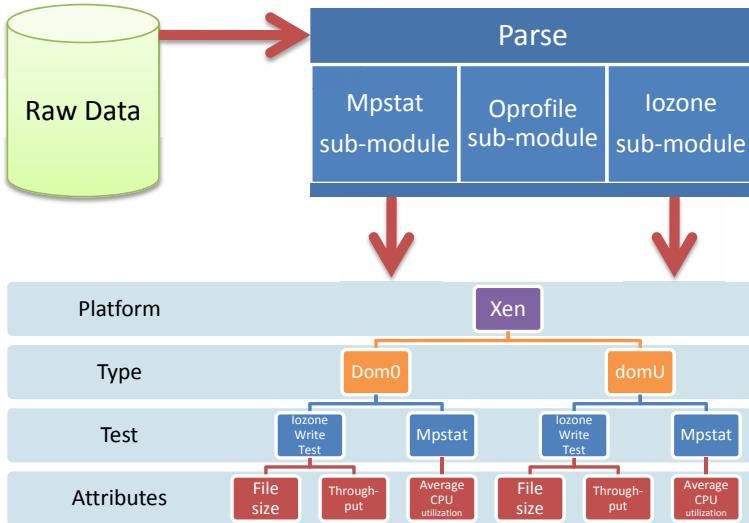


Fig. 3. The parsing sub-module of the *mine* component: *mine* deals with the parsing and presentation of the data gleaned by *batch*. The *parse* sub-module creates attributes dynamically from the data stored by sqlite. It then stores the parsed data back into temporary tables and passes it to one of several modules designed for aiding data discovery.

Mine aids in organizing, presenting, and analyzing the data collected by *batch*. A single test is represented as a datapoint containing various attributes pertinent to the application it involved. For example, a disk write test may record the file size, record size, operating system version, current disk scheduler, throughput, and so on. For the same test, the profiler employed may also record CPU and memory usage statistics. The only requirement for collecting the data is that the attributes, when parsed, have take the form of a few standard data types. An example of this parsing process is depicted in Figure 3. When aggregated into a single datapoint and combined with a multitude of other tests, we can begin to mine the data for trends and produce detailed statistics. Given a series of tests, for instance, we may choose to plot CPU usage and throughput versus varying record sizes. We can break this down even further by separating tests on the type of disk scheduler used. We can easily aggregate different batches in this way, since the batch itself is represented as just another attribute. All of these options have the same effect of reducing the dimensionality of the data to a point where results can be analyzed statistically or graphically plotted. We find this way of representing complex data to be very flexible. For example, if we switch from *vmstat* to *iostat* in our testing, we do not have to worry about creating new rules governing the new attributes that can be gleaned from this test. We can just record the attributes, and using the extensibility of the model, we can then choose the new attributes during the mining process. An example plotting wizard session is shown in Figure 4.

```

* | test name | size
1 kvmFRead-2_1025_4 | 1025kB
2 kvmFReadNoBuf-2_1025_4 | 1187kB
3 kvmFReadNoPV-2_1025_4 | 1033kB
4 kvmFReadPV-2_1025_4 | 1042kB
5 xenRead-2_1025_4 | 1091kB
6 xenReadNoBuf-2_1025_4 | 1306kB
7 xenReadRecordsSize-2_1025_4 | 1544kB

which files? Enter numbers (ex. 1 3 4): 5
How many subplots? 1
Specify custom x range (y/n)? n
Setting up ./files/xenRead-2_1025_4...
 1 domo <type 'list'>
 2 domU <type 'list'>
which to print? Enter numbers (ex. 1 3 4): 1 2
 1 kB <type 'str'>
 2 wa <type 'float'>
 3 bo <type 'float'>
 4 bi <type 'float'>
 5 swpd <type 'float'>
 6 free <type 'float'>
 7 in <type 'float'>
 8 cs <type 'float'>
 9 r <type 'float'>
10 id <type 'float'>
11 recLen <type 'str'>
12 sy <type 'float'>
13 b <type 'float'>
14 cache <type 'float'>
15 us <type 'float'>
16 si <type 'float'>
17 throughput <type 'str'>
18 so <type 'float'>
19 buff <type 'float'>
x y1 y2 y3 ...? Enter numbers (ex. 1 3 4): 1 17
x ranges from 2048 to 1048528.
Another plot from same file (y/n)? n

```

Fig. 4. An example wizard session for the mine tool. Tests are selected, parsed, and mined according to the specific traces the user wishes to see. At the end of the wizard, a graph is automatically generated. The graph generated by this session would look similar to the first subplot of Figure 6.

2.1 Implementation

We decided to use Python as our sole language, due to its object-oriented features and rising ubiquity as a high-level scripting language. In order to provide scalability and functionality on distributed systems, we disperse small TCP servers that coordinate jobs among the nodes. We use the *pylab* package, specifically *matplotlib*, for our plotting utilities. For our backend, we obtain lightweight database support with the python-supported *sqlite* standalone database system.

Several plug-in profiling modules have already been created for use with ExPerT, with several more forthcoming. The current modules are:

- Several tools from the sysstat suite [6], including mpstat and iostat
- Vmstat, a virtual memory monitoring tool [5]
- OProfile, a system-wide statistical profiler for Linux [1]

The only precondition for integration within the framework is that it follow the *prerun*, *run*, *postrun* format. Table II shows typical ways that these tools can be fit into this model. The actual configuration is specified by two xml files: one containing the node login details for the tested machines and one containing the *batch* module configuration. An example of latter file is shown in [5]. The *profile* and *bench* modules are specified by assigning tags, which serves to specify the tool involved and the node on which it is to run during the test. Since there are no *prerun* or *postrun* requirements, these tags can be left out. The *node* tag

```

<batch type='kvm'>
  <test>
    <profile>
      <title>virtual test</title>
      <name>vmstat</name>
      <args>1</args>
      <node>oscarnode1.qcluster</node>
    </profile>
    <bench>
      <title>virtual test</title>
      <name>iozone</name>
      <node>virtnode1</node>
    </bench>
  </test>
  <args>
    <bench argsTemplate=' -i 1 -s 4dm -r 512k -n -f /tmp/io.tmp'>
      <start>2</start>
      <stop>1024</stop>
      <step>2</step>
    </bench>
    <profile argsTemplate='1' />
  </args>
</batch>

```

Fig. 5. An example xml file for running a series of batch tests. Note that the *node* tag can be a remote machine.

Table 1. An example of different module configurations

Application	Thread Type	Prerun	Run	Postrun
Iozone	master	dd if=/dev/random of=io.tmp bs=1024 count=1024	iozone -i 1 -n -s 1024k -r 512k -f io.tmp	(none)
mpstat	slave	(none)	mpstat 1	kill mpstat
vmstat	slave	(none)	vmstat 1	kill vmstat
OProfile	slave	opcontrol -init opcontrol -reset	opcontrol -start	opcontrol -shutdown opreport -l

includes the name or IP of the machine. Leaving off this tag will run the test on localhost. The batch tags use C's *printf* syntax and set up how the tests are to iterate.

3 Experiments in Disk IO with KVM and Xen

Though ExPerT can be used for any distributed profiling applications, we decided to use ExPerT in our initial experiments to probe into the performance characterization of an ever-growing popular distributed system: virtual systems. Since virtual systems are typically made to communicate through the network stack, ExPerT can be naturally used for these types of applications without any modification. We decided to look into virtual disk I/O performance because it remains a relatively unstudied area, in contrast to virtual CPU performance or virtual network I/O performance. It has been shown that when running CPU-intensive benchmarks, performance nearly parallels that of the native execution environment. Virtual I/O performance is a difficult problem due to the complexities of page tables, context switching, and layered execution paths. Furthermore,

disk I/O is a vital operation for operations such as live migration, checkpointing, and other fault tolerant or data redundant strategies.

3.1 Experiments with ExPerT

Using ExPerT, we conducted a series of virtual disk I/O benchmarking experiments. Our hardware is composed of a Dell OptiPlex 745, equipped with an Intel Core 2 Duo 6400 with VT enabled, 2 Gigabytes of RAM, and a Barracuda 7200.10 SATA hard drive. Our software stack consists of the the Kernel-based Virtual Machine (KVM) [12] with Ubuntu 8.04 and Xen 3.2 [13]. For all our experiments, we used the Iozone application as our workloads.

KVM Disk I/O Paravirtualization. Beginning with the Linux kernel version 2.6.25, the *virtio-blk* module for paravirtual block I/O was included as a standard module. Using KVM (KVM-72), we looked for any performance benefits under Ubuntu 8.04 with a 2.6.26 kernel using this *virtio-blk* module. We did an extensive series of tests with ExPerT and found little performance difference with Iozone. A sample performance graph is shown in Figure 6 for the Read test. *wa*, *sy*, and *us* refer to I/O wait, system, and user percent processor utilization, respectively. *cs* is the number of average context switches per second. We did

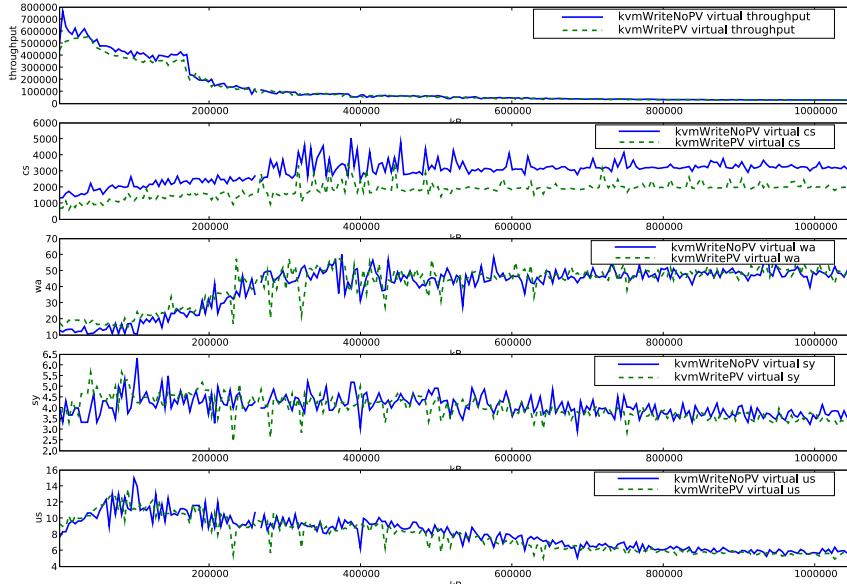


Fig. 6. KVM Paravirtualization: Tests with and without the paravirtual block I/O modules. Other than a small difference in context-switching performance, we could not see any real performance difference. We attribute this lack of improvement to the fact that the module is not very mature and has yet to be optimized.

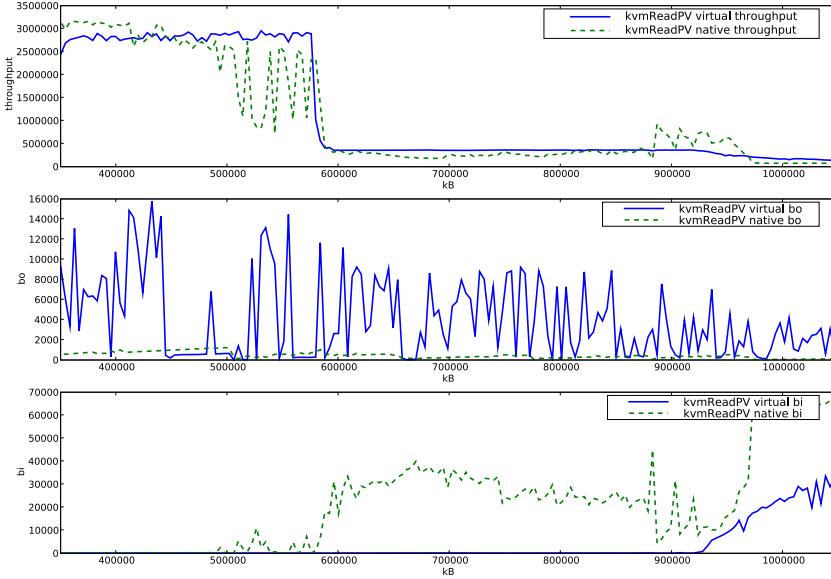


Fig. 7. KVM Guest Execution vs. Native Execution: Benchmarking and profiled memory statistics. The tags “bo” and “bi” represent the number of blocks per second (average) written out or read in from a block device (hard disk). The block size is 1024 bytes. The host is Ubuntu 8.04 Linux 2.6.25.9, and the guest is Ubuntu 8.04 Linux 2.6.26. Paravirtual I/O drivers were used.

find a small improvement in the number of average context switches per second reported by *vmbench*. We believe the module shows promise but it simply too immature and needs further optimization.

KVM Memory Management. Shown in Figure 7, we did a file read batch and looked at the average blocks per second read in from the disk and the average blocks per second written out to the disk. Since these statistics show the actual hard disk usage, we can immediately deduce the caching properties of the system. The results clearly show that for the native reads, the blocks read in from the disk start to increase about as the file size increases beyond the available free RAM. Another increase can be seen when nearing 1 Gigabyte, which correlates with a drop in overall throughput. We can also see the caching mechanisms of KVM at work, only requiring significant reading from the disk at just over 900 Megabytes. Also, note that due to the copying between the guest and KVM, there is significant blocks out per second for all virtual tests.

KVM and OProfile. We also ran tests with OProfile integrated into the framework while running *Iozone*. We choose to look at the number of data TLB misses for KVM guests and for the native systems running the *Iozone* benchmark with

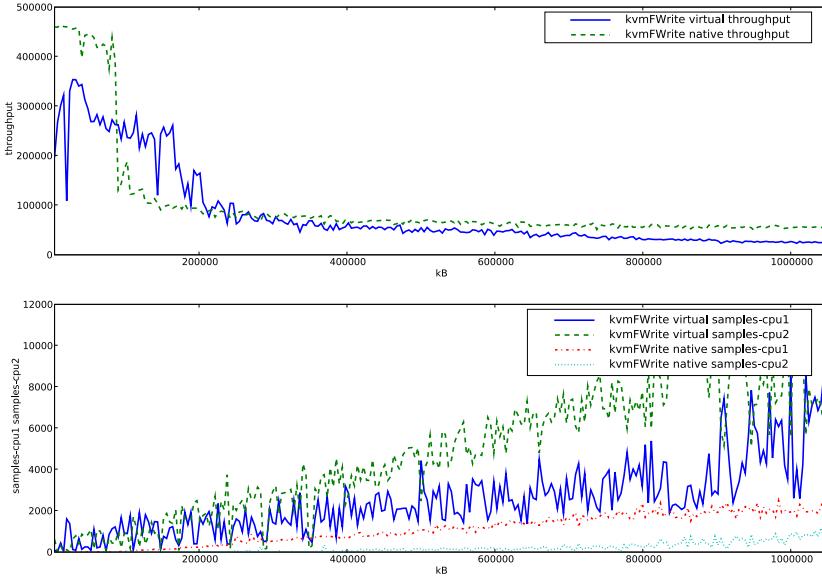


Fig. 8. KVM and OProfile: Detailed profiling statistics from OProfile while running Iozone. We chose to display the DTLB misses in this figure. The y-axis of the bottom subplot is in 1000's of misses.

the FWrite test. In the past, this has shown to be a source of significant overhead for virtual machines [24]. As expected, the number of DTLB misses grew substantially more when virtualized than the native testing. It is also curious that the DTLB misses were dominated by the first CPU. We found this to be the case across most OProfile tests.

KVM and Xen Performance. Finally, we detailed comparative performance tests of Xen and KVM. Figure 9 shows the results of three tests: FRead, FWrite, and Random, a random mix of reads and writes. Xen dom0 showed peculiarly good performance, even besting native performance. We expected good performance from Xen since it employs mature, paravirtualized I/O, but the performance is much better than we were expecting. The only type of disk operation where it wasn't the best was the 175-300 Megabyte range for file writing. Other than the dom0 outlier datapoints, we found Xen and KVM to be fairly comparable overall. The most drastic performance difference seem to come from differences in caching strategies, as evidenced by the varying plateau-like descents of the throughputs for the different file operations.

Though a more thorough analysis of this subject is still warranted, such analysis is beyond the scope of this paper, which it to present the toolkit and its capabilities, and provides the subject for future work.

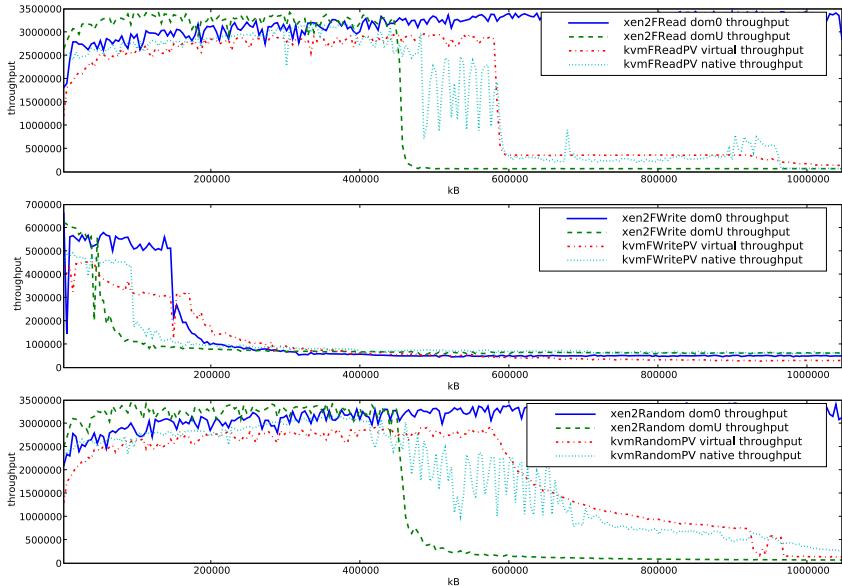


Fig. 9. KVM vs Xen: Read and Write operations for KVM, dom0, and domU

4 Related Work

Virtual disk I/O performance studies have been carried out before [15] as well as research dealing with virtualized storage [16], and though the focus of our paper is primarily to describe our framework, our case study deals closely with other notable attempts to characterize virtual disk I/O, including recent comparisons between KVM and Xen [17].

In the area of I/O benchmarking, there exist many different strategies and applications. For virtual systems, the SPEC virtualization committee has been working on a new standard benchmarking application [18]. Other virtual benchmarks include vmbench [19] and VMmark [20], a benchmarking application made by VMware. There is a wealth of research and toolkits on general I/O benchmarking techniques as well [21][22]. Finally, there exist anomaly characterization methods used in conjunction with I/O benchmarking for the purposes of pinpointing bugs and finding areas of improvement [23].

ExPerT is not a benchmark itself, but a characterization framework that integrates benchmarking, experimental design, and standard analysis. The most closely related projects to ExPerT include the Phoronix Test Suite, a general benchmarking framework [24], and Cbench, a cluster benchmarking framework [25]. The Phoronix Test Suite is limited in representation and functionality and relies heavily on the idea of community results distributed via the web. Some of the ideas behind Cbench are shared with ExPerT; Cbench places an emphasis on supporting an array of various benchmarking applications, and to a certain

extent, provides a common way to display results. In contrast, ExPerT centers around detailed workload characterization, standard analysis tools, and support for dynamic reconfiguration. Thus, the functionality of Cbench is encapsulated by the workload characterization portions of ExPerT, allowing Cbench to be used as the workloads component itself.

5 Conclusion

ExPerT is a broad, versatile tool for I/O characterization in distributed environments, facilitating benchmarking and profiling, and analysis and presentation. ExPerT consists primarily of two main modules: *batch* for automated batch testing and *mine* for automated data discovery, analysis, and presentation. Following the design principles of modularity, extensibility, and consistency, we have built ExPerT as a tool that eases the task of analysis and characterization of I/O on distributed systems. We have designed ExPerT to be a natural fit for large distributed systems, transparently employing remote connections to distribute jobs across several machines and in parallel. In our sample comparison study of virtual disk I/O in typical virtual distributed systems (KVM and Xen), ExPerT proves to be efficient to gather, sift, and analyze the large quantities of information that result from extensive testing. We believe ExPerT shows much promise as an I/O characterization and analysis framework for distributed environments. For more information, the ExPerT tool is available on the website <http://iweb.tntech.edu/hexb/ExPerT.tgz>.

Acknowledgment

This research was supported by the U.S. National Science Foundation under Grant No. CNS-0720617 and by the Center for Manufacturing Research of the Tennessee Technological University.

References

1. OProfile: A system-wide profiler for linux, <http://oprofile.sourceforge.net>
2. Sandmann, S.: Sysprof: a system-wide linux profiler, <http://www.daimi.au.dk/sandmann/sysprof>
3. Boehm, H.J.: The qprof project, <http://www.hpl.hp.com/research/linux/qprof/>
4. Graham, S.L., Kessler, P.B., Mckusick, M.K.: Gprof: A call graph execution profiler. In: SIGPLAN 1982: Proceedings of the 1982 SIGPLAN symposium on Compiler construction, pp. 120–126. ACM, New York (1982)
5. Vmstat: Vmstat man page, http://www.linuxcommand.org/man_pages/vmstat8.html
6. Godard, S.: Sysstat utilities homepage. <http://pagesperso-orange.fr/sebastien.godard/>
7. Iozone: Iozone file system benchmark, www.iozone.org

8. McVoy, L., Staelin, C.: lmbench: portable tools for performance analysis. In: ATEC 1996: Proceedings of the annual conference on USENIX Annual Technical Conference, Berkeley, CA, USA, pp. 23–23. USENIX Association (1996)
9. Coker, R.: The bonnie++ file-system benchmark,
<http://www.coker.com.au/bonnie++/>
10. Tirumala, A., Qin, F., Ferguson, J.D.J., Gibbs, K.: Iperf-the tcp/udp bandwidth measurement tool, <http://dast.nlanr.net/Projects/Iperf/>
11. Dbench: The dbench benchmark, <http://samba.org/ftp/tridge/dbench/>
12. Kivity, A., Kamay, Y.D., Laor, U.L., Liguori, A.: kvm: the linux virtual machine monitor. In: OLS 2007: Proceedings of the 2007 Ottawa Linux Symposium, Ottawa, Ontario, Canada, pp. 225–230. USENIX Association (2007)
13. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: SOSP 2003: Proceedings of the nineteenth ACM symposium on Operating systems principles, pp. 164–177. ACM, New York (2003)
14. Menon, A., Santos, J.R., Turner, Y., Janakiraman, G.J., Zwaenepoel, W.: Diagnosing performance overheads in the xen virtual machine environment. In: VEE 2005: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments, pp. 13–23. ACM, New York (2005)
15. Ahmad, I., Anderson, J., Holler, A., Kambo, R., Makhija, V.: An analysis of disk performance in vmware esx server virtual machines (October 2003)
16. Huang, L., Peng, G., cker Chiueh, T.: Multi-dimensional storage virtualization. In: SIGMETRICS 2004/Performance 2004: Proceedings of the joint international conference on Measurement and modeling of computer systems, pp. 14–24. ACM, New York (2004)
17. Deshane, T., Shepherd, Z., Matthews, J., Ben-Yehuda, M., Shah, A., Rao, B.: Quantitative comparison of xen and kvm. In: Xen Summit, Boston, MA, USA, June 2008, pp. 1–2. USENIX Association (June 2008)
18. Standard Performance Evaluation Corporation: Spec virtualization committee, <http://www.spec.org/specvirtualization/>
19. Moeller, K.T.: Virtual machine benchmarking (April 17, 2007)
20. Makhija, V., Herndon, B., Smith, P., Roderick, L., Zamost, E., Anderson, J.: Vmmark: A scalable benchmark for virtualized systems (2006)
21. Joukov, N., Wong, T., Zadok, E.: Accurate and efficient replaying of file system traces. In: FAST 2005: Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies, Berkeley, CA, USA, p.25. USENIX Association (2005)
22. Anderson, E., Kallahalla, M., Uysal, M., Swaminathan, R.: Buttress: A toolkit for flexible and high fidelity i/o benchmarking. In: FAST 2004: Proceedings of the 3rd USENIX Conference on File and Storage Technologies, Berkeley, CA, USA, pp. 45–58. USENIX Association (2004)
23. Shen, K., Zhong, M., Li, C.: I/o system performance debugging using model-driven anomaly characterization. In: FAST 2005: Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies, Berkeley, CA, USA, p. 23. USENIX Association (2005)
24. PTS: Phoronix test suite, www.phoronix-test-suite.com
25. Cbench: Scalable cluster benchmarking and testing,
<http://cbench.sourceforge.net/>

Grouping MPI Processes for Partial Checkpoint and Co-migration

Rajendra Singh and Peter Graham*

Dept. of Computer Science
University of Manitoba
Winnipeg, MB, Canada, R3T 2N2
{rajendra,pgraham}@cs.umanitoba.ca

Abstract. When trying to use shared resources for parallel computing, performance guarantees cannot be made. When the load on one node increases, a process running on that node will experience slow down. This can quickly affect overall application performance. Slow running processes can be checkpointed and migrated to more lightly loaded nodes to sustain application performance. To do this, however, it must be possible to; 1) identify affected processes and 2) checkpoint and migrate them independently of other processes which will continue to run.

A problem occurs when a slow running process communicates frequently with other processes. In such cases, migrating the single process is insufficient. The communicating processes will quickly block waiting to communicate with the migrating process preventing them from making progress. Also, if a process is migrated “far” from those it communicates with frequently, performance will be adversely affected.

To address this problem, we present an approach to identify and group processes which we expect to be frequent communicators in the near future. Then, when one or more process is performing poorly, the entire group is checkpointed and co-migrated. This helps to improve overall application performance in shared resource environments.

1 Introduction

Using idle compute resources is cost-effective and systems like Condor have successfully exploited such resources in limited contexts (e.g. parameters sweep studies). Increasingly, networks in large organizations are becoming more capable and, when combined with latency tolerance mechanisms, can now provide an attractive platform for running message passing parallel programs as long as the inter-process communication is not intensive.

In environments where machines are shared, however, load guarantees cannot be made. If one or more machines become overloaded it will decrease application performance. This provides a strong motivation to be able to checkpoint and migrate the affected processes to new machines. Such *performance-driven* migration should not involve the entire set of application processes as this would be wasteful both in terms of lost progress and overhead (from migrating processes).

* Supported by the Natural Sciences and Engineering Research Council of Canada.

We previously extended LAM/MPI to provide *partial* checkpoint and migration [1]. Our system checkpoints only those MPI processes that *need* to migrate due to overloading on their host nodes. For such partial checkpoint and co-migration to be effective, however, inter-process communication patterns must be considered which is the focus of this paper. Our prototype instruments the LAM/MPI Request Progression Interface (RPI) to efficiently gather pair-wise inter-process communication events. This data forms the basis on which pattern discovery can be done to determine inter-process communication patterns. These patterns are then used to predict the communication patterns expected in the near future, following a checkpoint event. The predicted patterns, in turn, provide the information needed to determine groups of processes that are expected to communicate frequently with one another. If one process in such a group needs to be migrated then all the processes in that group are checkpointed and co-migrated to a new set of lightly-loaded nodes. Thus, we ensure that these processes will be located near each other and thus continue to run effectively.

The rest of this paper is organized as follows. In Section 2 we briefly overview our prototype system for partial checkpoint and migrate. We then describe our approach to discovering communication patterns in Section 3. Section 4 explains how we predict groups of future frequently communicating processes using the discovered patterns. Some results of using our prediction algorithms applied to synthetic communications data are given in Section 5. We then briefly review related work in Section 6. Finally, in Section 7, we present some conclusions and discuss our planned directions for future work.

2 Our Partial Checkpoint/Migrate System

We have developed a prototype system for *partial* checkpoint and migration of MPI applications in a LAM/MPI environment that builds on LAM/MPI's SSI (System Software Interface) plugin architecture and the Berkeley Lab Checkpoint and Restart (BLCR) facility [2]. Our system [1] enables checkpoint, migration and restart of only those processes that have been impacted by overloaded compute nodes. A fundamental challenge in doing this is to be able to allow unaffected processes to continue their execution while the affected processes are being checkpointed, migrated and restarted. We have modified the LAM/MPI code base slightly to allow this and also to efficiently gather inter-process communication information on a pair-wise basis.

In our earlier work, we sampled communication information periodically and maintained summary information over multiple time scales (e.g. 1, 10 and 100 time units). This information was then weighted in various ways to try to determine the frequently communicating process groups. In our initial implementation we naively weighted recent information over older information and defined a simple threshold for deciding whether or not two processes were communicating frequently enough that they should be checkpointed together. Not surprisingly, the effectiveness of this simple grouping strategy was limited. In this paper we present new approaches to predicting future communication patterns that allow

us to better group frequently communicating processes for checkpointing and co-migration and we assess the effectiveness of the groupings.

3 Discovering Communication Patterns

MPI applications exhibit varying communication patterns over time. For example, processes in an FFT computation will communicate with other processes as determined by the “butterfly” exchange pattern inherent in the algorithm. A process P_i will communicate with different processes depending on the phase of the computation. We need to be able to discover and group frequently communicating processes accordingly. Naturally, applications where all processes communicate frequently will be unable to benefit from our work.

We predict future communication patterns based on previously observed patterns. Using the predicted patterns, we identify groups of processes that communicate frequently so that, when necessary, they can be checkpointed and co-migrate together. The basis of any prediction scheme relies on our ability to detect communication patterns using the data on pair-wise communication events collected by our system. We considered three possible approaches to identifying recurring patterns in the data: machine learning (e.g. the construction and use of Hidden Markov Models [3]), data mining techniques (e.g. sequential mining [45]) and various partial match algorithms.

Knowing that communication patterns between parallel processes sometimes vary and wanting to be able to reason about the pattern matching that is done so we could “tune” our approach, we decided to explore partial match algorithms. The TEIRESIAS [6] pattern matching algorithm is used for a number of bioinformatics problems related to sequence similarity – how “close” one sequence is to another. Of interest to us, TEIRESIAS is able to find long recurring patterns in sequences and is also able to do this subject to slight differences in the patterns. Identifying these types of patterns is important to determining groups of communicating processes. Relatively long, repeating patterns of nearly identical pair-wise communication events form the basis for determining which groups of processes can be reliably expected to communicate with each other.

When applied to DNA nucleotide sequences, the input alphabet, Σ , used by TEIRESIAS would simply be: { A,C,T,G } and the recurring patterns discovered would be sequences of these symbols interspersed with some, relatively small, number of “don’t care” symbols representing minor variations in the repeating patterns. To use TEIRESIAS to discover patterns in inter-process communication we must define an alphabet based on the communications data. Each pair of processes is mapped to a unique symbol using the mapping function: $\Theta < p1, p2 > = (p1 \times \text{NumProcs}) + p2$, where $p1$ and $p2$ are the process ranks, and NumProcs is the number of MPI processes. TEIRESIAS can support up to $2^{31} - 1$ symbols and its running time is independent of the number of symbols.

The output from our implementation of TEIRESIAS is a list ordered by length of commonly occurring patterns in an input sequence of communication events. Each such pattern is accompanied by a list of offsets identifying where they

occur in the input sequence. This offset information is used to identify patterns immediately preceding where a checkpoint occurs. It is also useful for identifying points at which communication patterns change.

4 Predicting Communicating Groups of Processes

Given the frequently occurring patterns discovered by TEIRESIAS in the inter-process communication data we collect in our modified version of LAM/MPI, our goal is to be able to respond to a need to checkpoint and migrate by accurately predicting which processes should be grouped together based on expected inter-communication. This can be done in a number of ways, subject to certain characteristics of our proposed environment.

4.1 The Prediction Environment

TEIRESIAS, while efficient, like other pattern discovery techniques, can be computationally expensive for complex patterns and long sequences. The communication data collected by our instrumented LAM/MPI implementation is periodically copied asynchronously to the TEIRESIAS-based pattern discovery process which runs in parallel with the MPIRUN process that starts and coordinates the processes forming each running MPI application. Ideally, and not unrealistically, these two “control” processes will run on a very capable host system.

The collection, transfer and analysis of communication patterns occurs on a regular basis, defining fixed length “sampling periods”. The need to checkpoint processes due to low performance occurs at unpredictable times and therefore asynchronously with respect to the sampling periods. Predictions about communicating process groups in the current sampling period must be made based on patterns extracted from preceding sampling periods.

4.2 Inter-Process Communication Characteristics

Message passing parallel applications can exhibit very different communications behaviour and resulting inter-process communication patterns. Application communication behaviour may depend on the parallel programming style used, the characteristics of the data being processed or on specifics of the selected algorithm(s). In many cases, however, there is a good deal of regularity in communications that can be exploited to identify groups of communicating processes.

For example, in Master/Slave parallel programming, the pattern of communication between the master and slaves, and among slaves is normally recurring over time. A typical Master/Slave approach involves three phases: data distribution, computation and data collection. The master distributes data to the slaves/workers in equal chunks. The workers then work on the data and, if required, communicate with other worker processes to exchange information. After computing, the workers send results to the master. This entire process repeats.

The use of Finite Element Methods (FEMs) for heat transfer problems is a good example of a calculation leading to recurring communication based on the structure of the data being processed. In such problems, modeling of the physical state of the entire system is partitioned and solved in parts across parallel processors. To calculate the temperature in one partition, the computing process needs to know the temperature in its neighboring partitions. The structure of the data partitioning thus determines the pattern of inter-process communication. In this kind of computation, the pattern of communication recurs for each time step that the application is executing.

Communication patterns may also be algorithm induced. An example of such an application is the computation of the Fast Fourier Transform (FFT) using the well known butterfly approach. This example results in changing but regular patterns of inter-process communication.

4.3 Possible Types of Predictors

The space of possible predictors can be divided into three major types as shown at the top of Fig. 1. A general, application-independent predictor will be simple but may be unlikely to give consistently good results for a range of applications. Custom, application specific predictors offer the best potential for accuracy but will require significant effort from application programmers who will likely have to create the predictors themselves. Instead, the use of a small number of generally useful predictors for a range of application types may be possible. These “specific” predictors will, in some sense, be application dependent but in a very general way, based on an application’s inter-process communication characteristics (such as those described previously). Using such an approach, application developers will be able to easily pick from a small set of available predictors based on the high-level characteristics of their applications. In this paper, we explore some possible predictors for what are perceived as some typical application types and then assess their performance.

4.4 Our Initial Predictors

We began by implementing two experimental predictors: a *basic* predictor that attempted to predict process groups based only on very recent communication

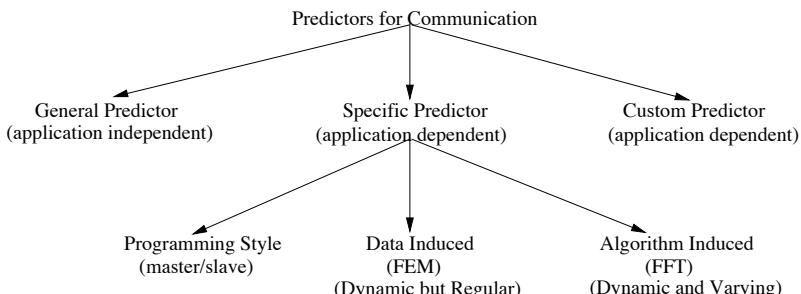


Fig. 1. Predictor Space

patterns, and a *historical* predictor which exploits information about communication patterns from earlier sampling periods, if available.

Our basic predictor selects the longest possible frequently occurring pattern in the most recent sampling period that has occurred at-least twice before the checkpoint arrival time and which is closest (in terms of time) to the time of checkpoint request. This is determined using the *offset* list and length of each pattern discovered by TEIRESIAS in that sampling period. If we are able to find a pattern meeting the criteria we designate it to be the basic prediction. This makes intuitive sense since it is the pattern of communication that appears to be on-going and is therefore, hopefully, likely to continue. If we don't find any pattern that meets these criteria then we simply pick the last pattern found from the previous sampling period and designate it as the basic prediction. This basic predictor serves as a baseline for the other predictors and is also used whenever the historical predictor cannot be used.

Our historical predictor was intended to address possible shortcomings of the basic predictor based on the fact that it only uses recent communication patterns to make a prediction. In the case where the application communication pattern is changing at the time of checkpoint, the immediately preceding patterns may not be the basis of a sound prediction. Our historical predictors identify patterns in the recent behaviour of the application and attempt to match these patterns against, possibly, older patterns stored from previous sampling periods. In this way, if the application is starting a new pattern of communication that has been seen earlier, we can make a more appropriate prediction based on the patterns from the earlier sampling period(s). This can be very effective for applications, like FFT, with varying but recurring communication patterns.

We implemented two versions of the historical predictor, one which makes a historical prediction only if there is an **exact** prefix match with a pattern from a previous period and the other which may accept a “close” historical match based on Levenshtein distance [7]. These are referred to as the *specific* and *Levenshtein* historical predictors, respectively.

Using the specific historical predictor, the pattern from the basic predictor is used as input and checked for an exact match against stored patterns seen in the past. If we find an exact match against patterns from one or more preceding sampling periods, then the pattern that occurred in the past immediately after the matched historical pattern becomes the predicted pattern. In the special case where the basic prediction exactly matches a pattern that occurred previously at the end of a sampling period, then we pick the first pattern that occurred in the next historical sampling period to be the predicted pattern. This process, naturally, chooses the communication pattern we expect to see next as the predicted pattern. If there is no match found then the specific historical predictor fails and the basic prediction is used.

The Levenshtein historical predictor is based on the concept of Levenshtein distance which simply reflects the number of mismatches between two compared patterns. The Levenshtein predictor works much as the specific predictor does except that it can make predictions when no exact match is found. We start with

the basic prediction and go through the stored historical patterns and compute the Levenshtein distance between the base prediction and each pattern. We select the pattern with the minimum Levenshtein distance (beneath an acceptability threshold) and designate its historical successor to be the predicted pattern.

4.5 Our Enhanced Predictors

We conducted a preliminary series of experiments (described in Section 5) using our initial prediction strategies. While the predictors generally performed reasonably well across a range of synthetically generated communication data, some aberrant behaviour was noted. We determined that the unexpected behaviour arose because TEIRESIAS was matching long patterns consisting of multiple repetitions of the same “fundamental” pattern. For example, given a frequently repeated inter-process communication pattern, A , TEIRESIAS was generating patterns such as AA , AAA , $AAAA$, etc. This was because the fundamental pattern (A in this case) occurred multiple times in immediate succession within a single sampling period. By favouring longer patterns, the fundamental pattern was ignored. Instead, a pattern close in size to the sampling period consisting of many repetitions of the fundamental pattern was selected. Unfortunately, this excessively long pattern did not reoccur frequently until the application had been running for some time. This resulted in poor performance when checkpoints were requested relatively early in an application’s execution.

To solve this problem, an enhanced prediction algorithm was created that explicitly checks for such repeated sequences, allowing it to identify the underlying fundamental patterns that are best used for prediction purposes. We use the Knuth-Morris-Pratt (KMP) pattern matching algorithm [8] to identify fundamental patterns by finding adjacent repeated patterns in the long patterns discovered by TEIRESIAS which consist solely of repetitions of a shorter pattern discovered by TEIRESIAS. The fundamental patterns identified in this way are then used in place of the original set of patterns found by TEIRESIAS.

5 Results

To assess the effectiveness of our predictors in a controlled environment, we chose to experiment using synthetically generated communications data. To ensure meaningful results, our synthetic data was created to reflect well-known communication patterns. This allowed us to start with clean and consistent patterns where we could more easily understand the behaviour of our predictors. We use the Levenshtein distance between our predicted pattern and the sequence seen in the synthetic data as our measure of goodness. We experimented with data reflecting the key communications characteristics of the three different sample application types discussed earlier: master/slave, FEM-like and FFT-like. Knowing that patterns in real communications data will vary, we introduced noise into and between the regular communications patterns for the FEM-like data to ensure that TEIRESIAS would still discover the necessary patterns. We created

hand-crafted data for a variety of “application sizes” and found consistent results independent of the number of processes. We also found that TEIRESIAS does tolerate slight differences in and between recurring patterns. We describe a subset of our experiments highlighting key results. In the figures that follow, ‘basic’ refers to predictions made using only recent communications behaviour, ‘historic’ refers to predictions using the exact match historic predictor and ‘levenshtein’ refers to predictions made using the approximate match (Levenshtein distance) historic predictor. Results for our enhanced predictors which use the KMP algorithm to identify fundamental patterns have a ‘(kmp)’ suffix. Result graphs are presented in 3D only where it improves feature identification.

Figure 2 shows the results using our initial prediction schemes (i.e. no identification of fundamental patterns) for an application with master/slave style communication pattern. The basic predictor, not unexpectedly, performs poorly, failing to correctly handle changes in communications patterns while the Levenshtein predictor performs surprisingly well. Of particular interest, however, is the drastic misprediction made by the historic predictor at, approximately, time 900 in this example run. This was caused because TEIRESIAS discovered a long pattern consisting of consecutive “fundamental” patterns which did not occur in the sampling period around time 900. This was the aberrant behaviour, described earlier, that lead us to develop our enhanced prediction schemes. Similar behaviour was also seen for other communication patterns (e.g. FFT-like).

As expected, our enhanced predictors perform significantly better than our initial ones, consistently providing high-quality prediction results for both master/slave and FFT-like communication patterns as can be seen in Fig. 3 and Fig. 4, respectively. Although graphs are only shown for the 8 process case, these results generalize to other problem sizes using more or less processes. They also are consistent with results for FEM-like communications with “noise”. The unexpectedly good performance of our original Levenshtein predictor can be explained by the closeness of the patterns seen in each sampling period to those

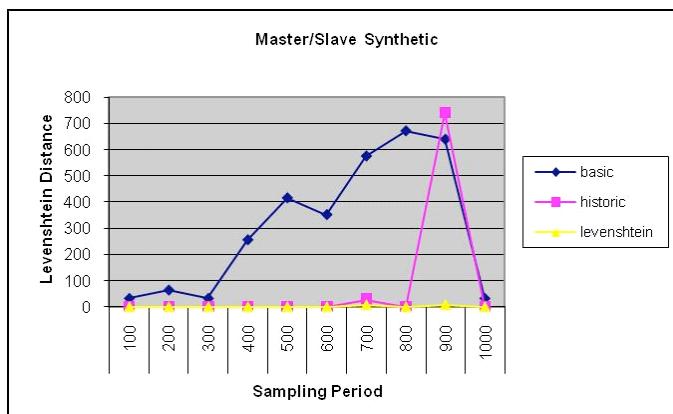


Fig. 2. Master/Slave Results for Original Predictors

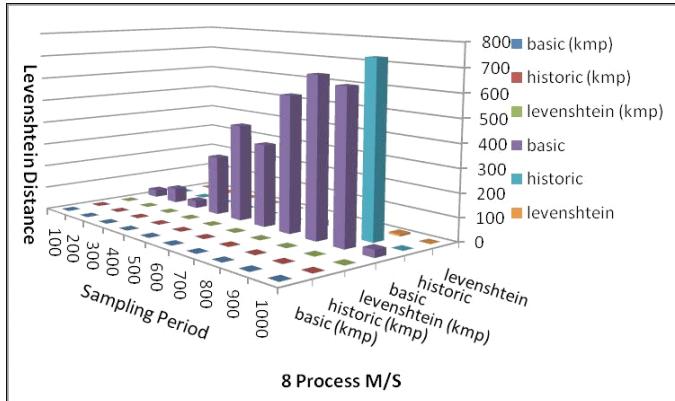


Fig. 3. Master/Slave Results for 8 Processes

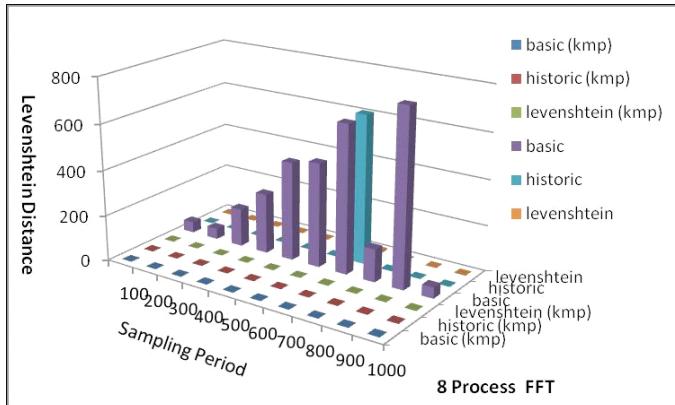


Fig. 4. FFT Results for 8 Processes

discovered by TEIRESIAS. We do not expect this performance to generalize well to real data and more complex communications patterns.

We also ran experiments to test the sensitivity of our predictors to the sampling period size. The results for the start of a four process master/slave communication pattern are shown in Fig. 5¹. Not surprisingly, our original predictors perform very poorly. It is also noteworthy that our enhanced predictors also show some variation. Choosing an appropriate sampling period size will likely be important to successful prediction for some patterns of communication.

Finally, we did some experiments where the large-scale characteristics of the various communication patterns were varied. For example, we increased the

¹ We use ribbon graphs for clarity. This is not intended to suggest continuous data.

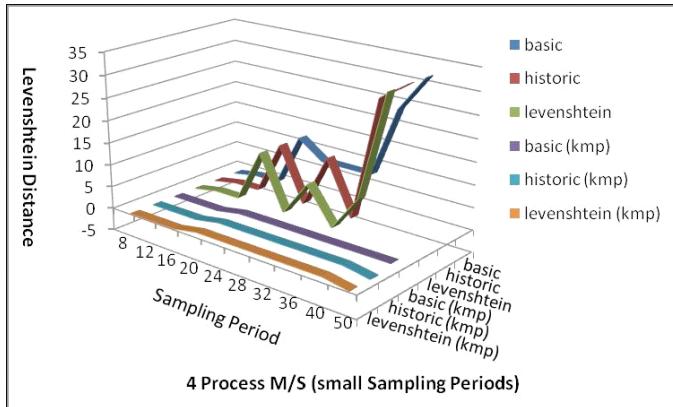


Fig. 5. Master/Slave Results with small sampling period for 4 Processes

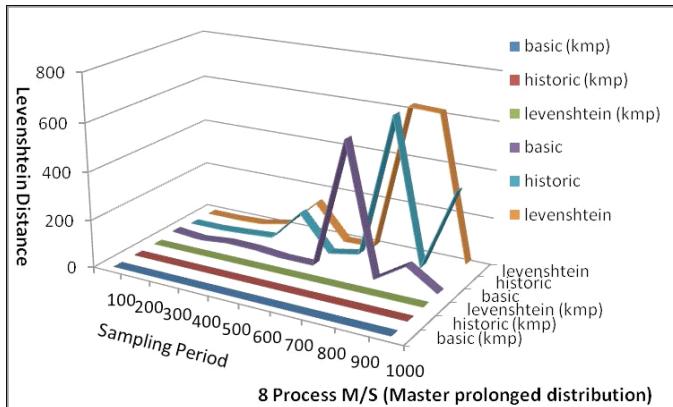


Fig. 6. Master/Slave Results with prolonged distribution phase for 8 Processes

duration of the data distribution phase for master/slave communications (refer to Fig. 6). In general, we found our enhanced predictors to be much more tolerant of a range of changes. This enforces our belief that there is value in determining fundamental patterns and using them for prediction.

6 Related Work

A number of systems have been built to add checkpointing to PVM, another message passing parallel programming system. These include Migratable PVM (MPVM) [9], and DynamicPVM [10] both of which support only full checkpointing. Vadhiyar et al [11] presented a migration framework designed for grid

environments and Huang, et al [12] described Adaptive MPI (AMPI) which provides “process” migration on top of and tightly coupled with Charm++. Most recently Wang, et al [13] have also developed a LAM/MPI partial checkpoint system. Unlike our system, theirs is focused on reliability rather than performance and does not permit any MPI processes to continue running during checkpoint and migration. In a large dedicated cluster environment (the focus of their work) this is not an issue since migrate and restart is fast and inter-process communication is uniform and efficient. In a shared environment such as the ones motivating our work, however, this is inefficient.

7 Conclusions and Future Work

We have developed a family of predictors that can be used to group frequently communicating MPI processes for checkpointing and co-migration to improve their performance on shared processors. Our predictors are based on (possibly refined) inter-process communication patterns discovered by the TEIRESIAS pattern discovery algorithm run against a sequence of pair-wise communication events captured by our LAM/MPI code enhancements. Our initial results using synthetic data that mimics known communication patterns are promising. Somewhat surprisingly, we saw relatively little difference between the various algorithms for different application types. This suggests that it may be possible to create a reasonably good “default” predictor that could be used where little is known about an application’s communications characteristics or where novice programmers cannot pick a specific predictor. Even if the consistent performance is partially an artifact of the synthetic data, we now have confidence that a range of message passing applications can be supported by only a few predictors.

We plan to explore several directions for future work. So far, we have only tested our predictors using synthetic data but we are now collecting actual inter-process communication data and will use this to verify the results obtained and to increase our confidence in our ability to handle small changes in communications behaviour that occur in real programs. With a sufficiently large set of test applications we should also be able to identify optimizations that will better suit particular application communications characteristics. We will explore this in an attempt to ensure consistent prediction accuracy. While TEIRESIAS outperforms other partial match algorithms using its “convolution” approach [6], its worst-case running time is still exponential for complex patterns with many variations. Therefore, it is not practical to run TEIRESIAS too frequently. Further, our predictor performance is sensitive to the size of the sampling periods. For the synthetic data used and for dozens of processes, our predictor runtimes were all sub-second but as the number of communication events increases so too will the runtime. We are interested in how to balance computational overhead and prediction accuracy and will do experiments to determine how few pattern discoveries we can do before missing important changes in inter-process communications for different types of applications. We will then attempt to create a mechanism to adapt the sampling period dynamically. We are also interested in

finding ways to predict when communication pattern changes are about to occur to avoid occasional mispredictions. Finally, while TEIRESIAS has performed well on the synthetic data, we are prepared to explore other algorithms to deal with any unanticipated problems as we move to real data.

References

1. Singh, R., Graham, P.: Performance Driven Partial Checkpoint/Migrate for LAM-MPI. In: 22nd International Symposium on High Performance Computing Systems and Applications (HPCS), June 2008, pp. 110–116 (2008)
2. Duall, J., Hargrove, P., Roman, E.: The design and implementation of berkeley lab's linux checkpoint/restart. Technical report, Berkley Labs LBNL-54941 (2002)
3. Brejova, B., Vinar, T., Li, M.: Pattern discovery: Methods and software. In: Krawetz, S.A., Womble, D.D. (eds.) *Introduction to Bioinformatics*, ch. 29, pp. 491–522. Humana Press (2003)
4. Srikant, R., Agrawal, R.: Mining Sequential Patterns: Generalization and Performance Improvement. In: Int'l Conf. Extending Database Technology, pp. 3–17 (1996)
5. Pei, J., Han, J.e.a.: PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth. In: Int'l Conf. Data Engineering, pp. 215– 226 (2001)
6. Rigoutsos, I., Floratos, A.: Combinatorial pattern discovery in biological sequences: The teiresias algorithm. *Bioinformatics* 14(1), 55–67 (1998)
7. Levenshtein, V.I.: Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. *Soviet Physics* 10, 707–710 (1966)
8. Knuth, D.E., Morris, J.H., Pratt, V.R.: Fast Pattern Matching in Strings. *SIAM Journal on Computing* 6(2), 323–350 (1977)
9. Casas, J., Clark, D., Konuru, R., Otto, S., Prouty, R., Walpole, J.: MPVM: A Migration Transparent Version of PVM. Technical report, CSE-95-002, 1 (1995)
10. Dikken, L., Linden, F.v.d., Vesseur, J., Sloot, P.: Dynamic PVM – Dynamic Load Balancing on Parallel Systems. In: *Proceedings Volume II: Networking and Tools*, pp. 273–277. Springer, Munich (1994)
11. Vadhiyar, S.S., Dongarra, J.J.: A Performance Oriented Migration Framework for the Grid. In: 3rd International Symposium on Cluster Computing and the Grid, Tokyo, Japan, May 12-15, 2003, pp. 130–137 (2003)
12. Huang, C., Lawlor, O., Kalé, L.V.: Adaptive MPI. In: Rauchwerger, L. (ed.) *LCPC 2003*. LNCS, vol. 2958, pp. 306–322. Springer, Heidelberg (2004)
13. Wang, C., Mueller, F., Engelmann, C., Scott, S.: A Job Pause Service under LAM/MPI+BLCR for Transparent Fault Tolerance. In: Proc. of the Parallel and Distributed Processing Symposium, pp. 1–10 (2007)

Process Mapping for MPI Collective Communications*

Jin Zhang, Jidong Zhai, Wenguang Chen, and Weimin Zheng

Department of Computer Science and Technology, Tsinghua University, China

{jin-zhang02, dijd03}@mails.tsinghua.edu.cn,
{cwg, zwm-dcs}@tsinghua.edu.cn

Abstract. It is an important problem to map virtual parallel processes to physical processors (or cores) in an optimized way to get scalable performance due to non-uniform communication cost in modern parallel computers. Existing work uses profile-guided approaches to optimize mapping schemes to minimize the cost of point-to-point communications automatically. However, these approaches cannot deal with collective communications and may get sub-optimal mappings for applications with collective communications.

In this paper, we propose an approach called OPP (Optimized Process Placement) to handle collective communications which transforms collective communications into a series of point-to-point communication operations according to the implementation of collective communications in communication libraries. Then we can use existing approaches to find optimized mapping schemes which are optimized for both point-to-point and collective communications.

We evaluated the performance of our approach with micro-benchmarks which include all MPI collective communications, NAS Parallel Benchmark suite and three other applications. Experimental results show that the optimized process placement generated by our approach can achieve significant speedup.

1 Introduction

Modern parallel computers, such as SMP (Symmetric Multi-Processor) clusters, multi-clusters and BlueGene/L-like supercomputers, exhibit non-uniform communication cost. For example, in SMP clusters, intra-node communication is usually much faster than inter-node communication. In multi-clusters, the bandwidth among nodes inside a single cluster is normally much higher than the bandwidth between two clusters. Thus, it is important to map virtual parallel processes to physical processors (or cores) in an optimized way to get scalable performance.

For the purpose of illustration, we focus on the problem of optimized process mapping for MPI (Message Passing Interface) applications on SMP clusters in this paper¹.

The problem of process mapping can be formalized to a graph mapping problem which finds the optimized mapping between the communication graph of applications

* This work is supported by Chinese National 973 Basic Research Program under Grant No. 2007CB310900 and National High-Tech Research and Development Plan of China (863 plan) under Grant No. 2006AA01A105.

¹ MPI ranks are used commonly to indicate MPI processes. We use terms *MPI ranks* and *MPI processes* interchangeably.

and the topology graph of the underlying parallel computer systems. Existing research work, such as MPI/SX [1], MPI-VMi [2] and MPIPP [3], addresses this problem by finding optimized process mapping for **point-to-point communications**. However, they all ignore **collective communications** which are also quite sensitive to process mapping.

In this paper, we propose a way to optimize process mapping for collective communications. Our approach called OPP is based on the observation that most collective communications are implemented through a series of point-to-point communications. Thus we can transform collective communications into a series of point-to-point communications according to their implementation in communication libraries. Then we can use the existing framework [3] to find out the optimized process mapping for whole applications.

The contributions of this paper can be summarized as follows:

- A method to find optimized mapping scheme for a given collective operation by decomposing it to a series of point-to-point communications.
- Integration of the above method with existing process mapping research work to obtain optimized process mapping for whole parallel applications which have both point-to-point communications and collective communications.
- We perform extensive experiments with micro-benchmarks, the NAS Parallel Benchmark suite (NPB) [4] and three other applications to demonstrate the effectiveness of our method.

Our paper is organized as follows, in Section 2 we discuss the related work, and Section 3 describes the brief framework about process placement mechanism. Section 4 introduces the method to generate communication topology of parallel applications. And the experimental environment and experimental results are shown in Section 5 and Section 6. We discuss the interaction between process mapping and collective communication optimization and propose an alternative way to deal with the process placement problem in Section 7. Conclusion is finally made in Section 8.

2 Related Works

Various process mapping approaches have been proposed to optimize the communication performance for message passing applications in SMP clusters and multi-clusters [5, 6, 3]. MPICH-VMi [2] proposes a profile-guided approach to obtain the application communication topology, and uses general graph partitioning algorithm to find optimized mapping from parallel processes to processors. But MPICH-VMi requires users to provide the network topology of the target platform. MPIPP [3] makes the mapping procedure more automatically by employing a tool to probe the hardware topology graph so that it can generate optimized mapping without users' knowledge on either applications or target systems. MPIPP also proposes a new mapping algorithm which is more effective for multi-clusters than previous work. Topology mapping on BlueGene/L has been studied in [7, 8] which describe a comprehensive topology mapping library for mapping MPI processes onto physical processors with three-dimensional grid/torus topology. However, none of the above work handles the problem of optimizing process mapping for collective communications, and may get sub-optimal process mapping results for applications with collective communications.

Much work has been done on optimized implementations of collective communications. For example, Magpie [9] is a collective communication library optimized for wide area systems. Sanders et al. [10], Sistare et al. [11] and Tippuraju et al. [12] discuss various approaches to optimize collective communication algorithms for SMP clusters. Some work focuses on using different algorithms for different message size, such as [13, 14]. None of previous work shows how it interacts with existing process placement approaches which are based on point-to-point communications and may also result in sub-optimal mappings.

Our work, to the best of our knowledge, is the first one to obtain optimized process mapping for applications with both collective communications and point-to-point communications.

3 The Framework of Process Placement

In this section, we illustrate the framework of process placement. In general, the process placement algorithm takes parameters from target systems and parallel applications, and outputs the process placement scheme, as illustrated in Figure 1.

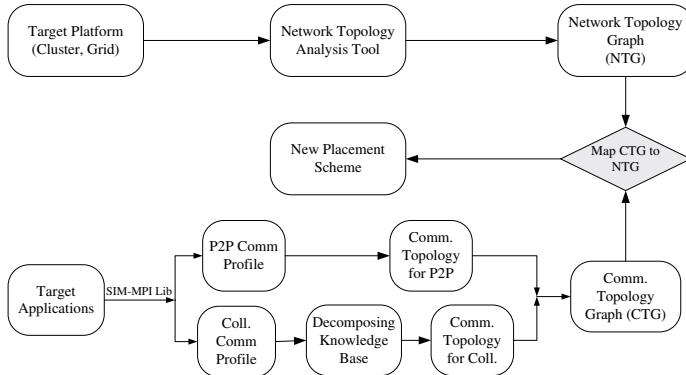


Fig. 1. The framework of our process placement method

Target systems are modeled with network topology graphs (NTGs) describing bandwidth and latency between processors (cores), which are obtained with a network topology analysis tool automatically. The tool is implemented with a parallel ping-pong test benchmark which is similar to the one used in MPIPP [3]. Two $M \times M$ matrices are used to represent network topology graphs, where M is the number of processor cores in the target system: (1) NTG_{bw} describes the communication bandwidth and (2) $NTG_{latency}$ describes the communication latency between each pair of two processor cores. We adopt the method used in MVAPICH [15] to measure and calculate latency and bandwidth between two processor cores.

Parallel applications are characterized with communication topology graphs (CTGs) which include both message count and message volume between any pair of MPI

ranks. We process point-to-point communications and collective communications separately. For point-to-point communications, we use two matrices, CTG_{p2p_count} and CTG_{p2p_volume} , to represent the number or aggregated volume of point-to-point communications between rank i and j in a parallel application respectively (Please refer to [3] for details). For collective operations, we propose a method to translate all collective communications into point-to-point communications, which will be described in detail in Section 4. Now assuming we have translated all collective communications into a series of point-to-point communications, we can generate the following two matrices CTG_{coll_count} and CTG_{coll_volume} in which element (i, j) represents the number or volume of **translated** point-to-point communications from **collective communications** between rank i and j respectively. Then the communication topology of the whole application can be represented by two matrices which demonstrate message count and message volume for both collective and point-to-point communications:

$$CTG_{app_count} = CTG_{coll_count} + CTG_{p2p_count}$$

$$CTG_{app_volume} = CTG_{coll_volume} + CTG_{p2p_volume}$$

We feed the network topology graphs and communication topology graphs to a graph partitioning algorithm to get the optimized process placement. In our implementation, we use the heuristic k-way graph partitioning algorithm proposed in [3] which gives a detailed description for its implementation and performance.

4 Communication Topology Graphs of Collective Communications

In this section, we introduce our approach to decompose collective communications into point-to-point communications. We first use *MPI_Alltoall* as a case study, then we show the construction of Decomposition Knowledge Base (DKB) which can be employed to transform every MPI collective communications into point-to-point communications.

4.1 A Case Study: *MPI_Alltoall*

One implementation of *MPI_Alltoall* is the Bruck Algorithm [16], as shown in Figure 2. At the beginning, rank i rotates its data up by i blocks. In each communication step k , process i sends to rank($i + 2^k$) all those data blocks whose k th bit is 1, receives data from rank($i - 2^k$). After a total of $\lceil \log P \rceil$ steps, all the data get routed to the right destination process. A final step is that each process does a local inverse shift to place the data in the right order.

For an *MPI_Alltoall* instance on 8 MPI ranks, we can decompose it in three steps as illustrated in Figure 2. Assuming the message size of each item is 10 byte, then *step 0* can be decomposed into 8 point-to-point communications whose message sizes are all 40 bytes. The second and third steps can also be decomposed into 8 point-to-point communications of 40 bytes respectively. Finally, we decompose the *MPI_Alltoall* into 24 different point-to-point communications. We aggregate the volume and number of messages between each pair of the MPI ranks and get the communication graphs (CTG_{coll_count} and CTG_{coll_volume}), which are first introduced in Section 3. Figure 3 shows the CTG_{coll_volume} for this *MPI_Alltoall* instance.

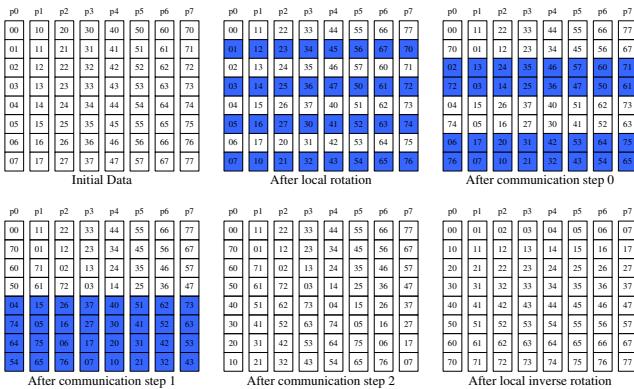


Fig. 2. Bruck Algorithm for $MPI_Alltoall$ with 8 MPI ranks. The number (ij) in each box represents the data to be sent from rank i to rank j . The shaded boxes indicates the data to be communicated in the next step.

P7	40	40	0	80	0	40	40	0
P6	40	0	80	0	40	40	0	40
P5	0	80	0	40	40	0	40	40
P4	80	0	40	40	0	40	40	0
P3	0	40	40	0	40	40	0	80
P2	40	40	0	40	40	0	80	0
P1	40	0	40	40	0	80	0	40
P0	0	40	40	0	80	0	40	40

Fig. 3. Decomposition results for Bruck Algorithm of *MPI_Alltoall* with 8 MPI ranks. The value of the block (i, j) represents the communication volume between the process i and the process j during the process of collective communications.

4.2 Decomposition Knowledge Base

The previous section shows how we can decompose *MPI_Alltoall* to point-to-point communications and generate its communication graphs. The same approach can be applied to other MPI collection communications too.

One of the challenges to decompose MPI collective communications is that they are implemented in different ways for different MPI libraries. What's more, even in the same MPI library, the algorithms used to implement a certain collective communication may depend on the size of messages and the number of processes.

To correctly identify implementation algorithms for each collective communication, we build a Decomposing Knowledge Base (DKB) which records the rules to map collective communications to its implementation algorithms. Through analyzing the MPICH-1.2.7 [18] code and MVAPICH-0.9.2 [15] code manually, we get the underlying

Name	MPICH-1.2.7	MVAPICH-0.9.2
Barrier	Recursive Doubling	Recursive Doubling
Bcast	Binomial Tree (SIZE<12288bytes or NP<8) Van De Geijn (SIZE<524288bytes, power-of-two MPI processes and NP≥8) Ring (the other conditions)	Binomial Tree (SIZE<12288bytes and NP≤8) Van De Geijn (the other conditions)
Allgather	Recursive Doubling (SIZE*NP<524288bytes and power-of-two MPI processes) Bruck (SIZE*NP<81920bytes and non-power-of-two MPI processes) Ring (the other conditions)	Recursive Doubling
Allgatherv	Recursive Doubling (SIZE*NP<524288bytes and power-of-two MPI processes) Bruck (SIZE*NP<81920bytes and non-power-of-two MPI processes) Ring (the other conditions)	Recursive Doubling (TOTAL_SIZE≤262144bytes) Ring(the other conditions)
Gather	Minimum Spanning Tree	Minimum Spanning Tree
Reduce	Rabenseifner (SIZE>2048bytes and OP is permanent.) Binomial Tree (the other conditions)	Recursive Doubling
Allreduce	Recursive Doubling (SIZE≤2048bytes or OP is not permanent.) Rabenseifner (the other conditions)	Recursive Doubling
Alltoall	Bruck (SIZE≤256bytes and NP≥8) Isend_Irecv (256bytes≤SIZE≤32768bytes) Pairwise Exchange (the other conditions)	Recursive Doubling (SIZE≤128bytes) Isend_Irecv (128bytes<SIZE<262144) Pairwise Exchange (the other conditions)
Scatter	Minimum Spanning Tree	Minimum Spanning Tree
Scatterv	Linear	Linear
Gatherv	Linear	Linear
Alltoally	Isend_Irecv	Isend_Irecv

Fig.4. Classify algorithms used in MPICH and MVAPICH. SIZE represents the communication size. NP represents the number of MPI processes. OP represents the operation used in MPI_Allreduce or MPI_Reduce and TOTAL_SIZE used for MPI_Allgatherv represents the sum of sizes received from all the other processes (Van de Geijn and Rabenseifner's algorithms are proposed in [13] and [17] respectively.).

collective communication algorithms listed in Figure 4. This table presents collective communication algorithms used in different collective communications. The DKB we built is a essentially similar table which can output the decomposition algorithms used by a specified MPI collective communication depending on the interconnect, the MPI implementation, message sizes and the number of processes. Our current DKB covers MPICH-1.2.7 and MVAPICH-0.9.2 for both Ethernet and Infiniband networks.

With the help of DKB, we can generate the communication topology graph of any collective communications, as shown in Figure 5. We input a collective communication to DKB, which include the type of the collective communication, Root_ID, Send_Buf_Size and Communicator_ID. DKB outputs the algorithm used in this

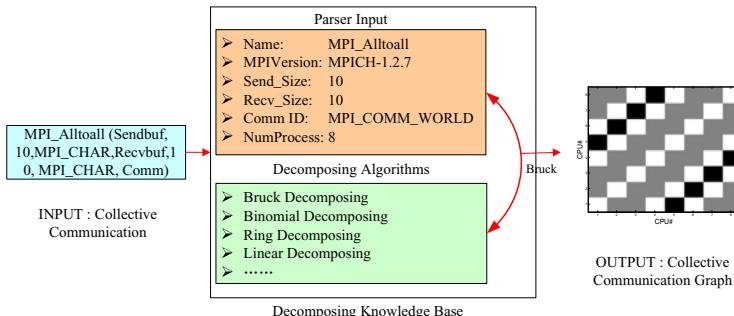


Fig.5. Usage of decomposing knowledge base

collective communication. We then decompose the collective communication into a series of point-to-point communications based on its implementation. These point-to-point communications can then be used to generate the communication topology graph of this collective communication. We process the collective communications one by one, and aggregate their communication topology graphs to obtain the communication topology graphs for all the collective communications (CTG_{coll_count} and CTG_{coll_volume} defined in Section 3) of the whole application. Then we can perform process mapping according to steps described in Section 3.

5 Experiment Platforms and Benchmarks

5.1 Experiment Platforms

We perform experiments on a 16-node cluster in which each node is a 2-way server with 4GB memory. The processors in this cluster are 1.6GHz Intel Xeon dual core processors with 4MB L2 cache. Linux 2.6.9 and MPICH-1.2.7 [18] are installed on each node. These nodes are connected by a 1Gbps Ethernet. Since there are 64 cores in the cluster, we execute up to 64-rank MPI applications and all applications are compiled with Intel compiler 9.0.

The cluster exhibits non-uniform communication cost between cores. We use our network analysis tool to get the network topology graphs for our experimental network platform. Figure 6 and Figure 7 show the latency and bandwidth between each pair of cores in this cluster. We see that communication inside a node is much faster than communication between nodes.

5.2 Benchmarks and Applications

We use the following benchmarks and applications to verify the effect of our optimized process placement scheme.

1. Intel MPI Benchmark (IMB) [19]

IMB is a micro-benchmark developed by Intel. We use it to verify that we can find optimized process placement for each MPI collective communication.

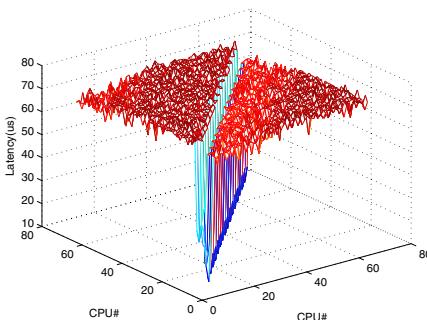


Fig. 6. Latency between cores

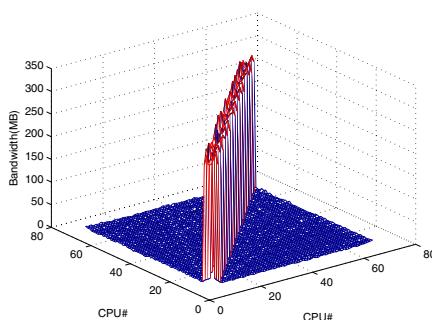


Fig. 7. Bandwidth between cores

2. NAS Parallel Benchmark (NPB) [4]

The NAS Parallel Benchmarks (NPB) are a set of scientific computation programs. In this paper, we use NPB 3.2 and Class C data set.

3. Three other applications

- ASP [9]

A parallel application which solves the all-pairs-shortest-path problem with the Floyd-Warshall algorithm. It has been integrated into Magpie [9].

- GE [20]

A message passing implementation of the Gauss Elimination. It is an efficient algorithm for solving systems of linear equations.

- PAPSM [21]

This is a parallel application which implements the realtime dynamic simulation of power systems. The application is implemented by a hierarchical Block Bordered Diagonal Form (BBDF) algorithm for power network computation.

6 Experiment Results and Analysis

In this section, we compare our process mapping approach *OPP* with two widely used process placement scheme in MPI, *block* and *cyclic* [22], as well as the most up-to-date optimized process mapping approach, *MPIPP* [3]. Each result is the average of five executions and normalized by the result of *block* scheme.

6.1 Micro Benchmarks

We use IMB² to evaluate how OPP outperforms *block* and *cyclic* for each individual collective communications. The result of MPIPP is not presented because MPIPP does not deal with collective communications at all.

Figures 8–15 exhibit the results of OPP, *block* and *cyclic* process placement for different collective communications on 64 MPI ranks. The results show:

- There are a few collective communications such as *Reduce*, *Allreduce*, on which the *block* scheme always outperforms the *cyclic* scheme, while there are a few others such as *Gather* and *Scatter*, on which the *cyclic* scheme always outperforms the *block* scheme.
- For some collective communications, such as *Bcast* and *Allgather*, neither the *block* nor *cyclic* scheme can always outperform the other because their relative performance depends on the size of messages.
- Our proposed approach, *OPP*, can always find the best placement scheme comparing to the *block* and *cyclic* for all collective communications on all message sizes.

² IMB does not provide the test for *MPI_Gather* and *MPI_Scatter*. We design a micro-benchmark to test them.

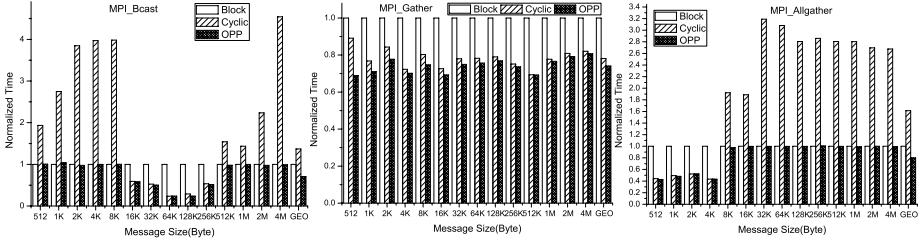


Fig. 8. Bcast (NP=64)

Fig. 9. Gather (NP=64)

Fig. 10. AllGather (NP=64)

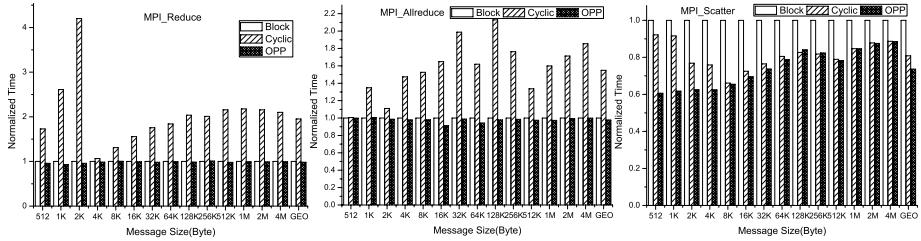


Fig. 11. Reduce (NP=64)

Fig. 12. Allreduce (NP=64)

Fig. 13. Scatter (NP=64)

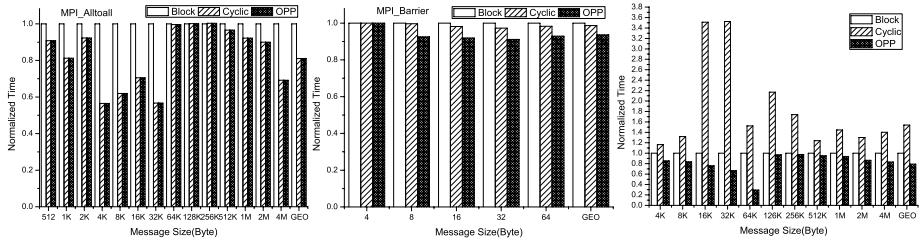


Fig. 14. Alltoall (NP=64)

Fig. 15. Barrier (NP=64)

Fig. 16. Allreduce (NP=33)

Non-Power-of-Two MPI Ranks. We take a further investigation on process placement for non-power-of-two MPI ranks. Some collective communication algorithms favor more on the situation of the power-of-two MPI ranks because they are symmetric in design, so their performance may degrade when the number of ranks is not power-of-two.

We show how OPP can improve performance in these situations. Due to space limitation, we only take *MPI_Allreduce* as an example. Figure 16 exhibits the result of *MPI_Allreduce* with 33 MPI ranks.

Figure 16 shows that *OPP* performs significantly better than the *block* and *cyclic* placement schemes for all tested message size. *OPP* is 20.4% better than the *block* placement and 53.6% better than the *cyclic* placement on average.

The result indicates that *OPP* has more benefits for applications with collective communications which require running with non-power-of-two MPI ranks.

6.2 NPB and Other Applications

In this section, we perform experiments with NPB and three parallel applications, ASP, GE and PAPSM as described in Section 5.2. The results are obtained with 64 MPI ranks except for PAPSM which only supports up to 20 MPI ranks. The results are shown in Table 1. For applications which only contain collective communications, such as *ft*, PAPSM and ASP, performance of MPIPP is not applicable since it can only find optimized placement for point-to-point communications.

By examining the results in Table 1, we can see that:

- For applications dominated by point-to-point communications, such as *bt*, *cg*, *sp* and *mg*, both MPIPP and OPP can get better performance than *block* and *cyclic* schemes. MPI and OPP are equally good for this category of applications.
- For applications that only contain collective communications, such as *ft*, *PAPSM* and *ASP*, MPIPP can not get optimized rank placement because it does not deal with collective communications. So we can only compare OPP with *block* and *cyclic* schemes. We see that OPP shows its capability to find optimized MPI rank placement scheme for this class of applications which can get up to 26% performance gain over the best of block or cyclic schemes.
- *IS* and *GE* are applications that have both point-to-point and collective communications, but are dominated by collective communications. MPIPP decides the MPI rank placement based on the point-to-point communication patterns and get suboptimal placement schemes. MPIPP is 6.0% and 5.1% worse than the best of *block* or *cyclic* scheme for *IS* and *GE* respectively. On the contrary, OPP can find optimized layout for both point-to-point and collective communications, which is 0.1% and 19% better in these two applications.

In summary, OPP shows that it can find optimized MPI process placement for all three classes of parallel applications.

Table 1. The execution time (in seconds) of NPB suite and three parallel programs with different placement schemes

Name	Block(s)	Cyclic(s)	MPIPP(s)	OPP(s)	Speedup of OPP vs. Block
bt.C.64	95.47	103.76	90.79	90.66	1.05
cg.C.64	64.14	89.03	62.3	62.3	1.03
ep.C.64	11.12	11.12	11.12	11.09	1.00
lu.C.64	69.32	74.39	68.72	68.72	1.01
sp.C.64	143.97	151.40	132.12	132.12	1.09
mg.C.64	9.56	9.12	9.11	9.10	1.06
is.C.64	12.59	12.14	12.87	12.13	1.04
ft.C.64	31.52	23.20	N.A.	22.89	1.38
PAPSM.20	15.78	19.45	N.A.	12.54	1.26
GE.64	20.83	25.14	21.89	17.48	1.19
ASP.64	55.93	50.46	N.A.	50.16	1.11

7 Discussion

An interesting issue is the interaction between MPI process placement and optimized collective communication implementation. In our current scheme, we first fix the collective communication implementation according to the MPI library, then perform process placement optimization based on this implementation. This is a reasonable choice if we wish our MPI process placement approach to be compatible with existing MPI libraries.

An alternative approach to deal with this problem is to fix the process placement based on the point-to-point communication pattern of a parallel application first, then determine the optimized collective communication implementation for the given process placement scheme. This approach has the potential of achieving better performance, but loses the compatibility because it only works with MPI libraries which are process placement aware. Nevertheless, we believe this is a promising way to go.

8 Conclusions

In this paper, we argue that it is an important problem to map virtual parallel processes to physical processors (or cores) in an optimized way to get scalable performance due to non-uniform communication cost in modern parallel computers. Existing work either determines optimized process mapping based on point-to-point communication patterns or optimizes collective communications only without awareness of point-to-point communication patterns in parallel applications. Thus they may all fall into sub-optimal placement results.

To solve the problem, we propose a method which first decomposes a given collective communication into a series of point-to-point communications based on its implementation in the MPI library that are used in the target machine. Then we generate the communication patterns of the whole application by aggregating all collective and point-to-point communications in it. We then use a graph partition algorithm to find optimized process mapping schemes.

We perform extensive experiments on each single MPI collective communications and 11 parallel applications with different communication characteristics. Results show that our method (OPP) can get best results in all cases, and perform significantly better than previous work for applications with both point-to-point and collective communications.

References

- [1] Colwell, R.R.: From terabytes to insights. *Commun. ACM* 46(7), 25–27 (2003)
- [2] Pant, A., Jafri, H.: Communicating efficiently on cluster based grids with MPICH-VMI. In: CLUSTER, pp. 23–33 (2004)
- [3] Chen, H., Chen, W., Huang, J., Robert, B., Kuhn, H.: MPIPP: an automatic profile-guided parallel process placement toolset for SMP clusters and multiclusters. In: ICS, pp. 353–360 (2006)
- [4] NASA Ames Research Center. NAS parallel benchmark NPB,
<http://www.nas.nasa.gov/Resources/Software/npb.html>
- [5] Phinjaroenphan, P., Bevinakoppa, S., Zeephongsekul, P.: A heuristic algorithm for mapping parallel applications on computational grids. In: EGC, pp. 1086–1096 (2005)

- [6] Sanyal, S., Jain, A., Das, S.K., Biswas, R.: A hierarchical and distributed approach for mapping large applications to heterogeneous grids using genetic algorithms. In: CLUSTER, pp. 496–499 (2003)
- [7] Bhanot, G., Gara, A., Heidelberger, P., Lawless, E., Sexton, J., Walkup, R.: Optimizing task layout on the Blue Gene/L supercomputer. IBM Journal of Research and Development 49(2-3), 489–500 (2005)
- [8] Yu, H., Chung, I., Moreira, J.: Topology mapping for Blue Gene/L supercomputer. In: SC, pp. 52–64 (2006)
- [9] Kielmann, T., Hofman, R.F.H., Bal, H.E., Plaat, A., Bhoedjang, R.: MagPIE: MPI’s collective communication operations for clustered wide area systems. In: PPOPP (1999)
- [10] Sanders, P., Traff, J.L.: The hierarchical factor algorithm for all-to-all communication (research note). In: Monien, B., Feldmann, R.L. (eds.) Euro-Par 2002. LNCS, vol. 2400, pp. 799–804. Springer, Heidelberg (2002)
- [11] Sistare, S., vande Vaart, R., Loh, E.: Optimization of MPI collectives on clusters of large-scale SMP’s. In: SC, pp. 23–36 (1999)
- [12] Tipparaju, V., Nieplocha, J., Panda, D.K.: Fast collective operations using shared and remote memory access protocols on clusters. In: IPDPS, pp. 84–93 (2003)
- [13] Barnett, M., Gupta, S., Payne, D.G., Shuler, L., van de Geijn, R., Watts, J.: Interprocessor collective communication library (InterCom). In: SHPCC, pp. 357–364 (1994)
- [14] Kalé, L.V., Kumar, S., Varadarajan, K.: A framework for collective personalized communication. In: IPDPS, pp. 69–77 (2003)
- [15] Ohio State University. MVAPICH: MPI over infiniband and iWARP, <http://mvapich.cse.ohio-state.edu>
- [16] Bruck, J., Ho, C., Upfal, E., Kipnis, S., Weathersby, D.: Efficient algorithms for all-to-all communications in multiport message-passing systems. IEEE Trans. Parallel Distrib. 8(11), 1143–1156 (1997)
- [17] Rabenseifner, R.: New optimized MPI reduce algorithm, <http://www.hlrs.de/organization/par/services/models/mpi/myreduce.html>
- [18] Argonne National Laboratory. MPICH1, <http://www-unix.mcs.anl.gov/mpi/mpich1>
- [19] Intel Ltd. Intel IMB benchmark, <http://www.intel.com/cd/software/products/asmo-na/eng/219848.htm>
- [20] Huang, Z., Purvis, M.K., Werstein, P.: Performance evaluation of view-oriented parallel programming. In: ICPP, pp. 251–258 (2005)
- [21] Xue, W., Shu, J., Wu, Y., Zheng, W.: Parallel algorithm and implementation for realtime dynamic simulation of power system. In: ICPP, pp. 137–144 (2005)
- [22] Hewlett-Packard Development Company. HP-MPI user’s guide, <http://docs.hp.com/en/B6060-96024/ch03s12.html>

Topic 2

Performance Prediction and Evaluation

Introduction

Thomas Fahringer^{*}, Alexandru Iosup^{*}, Marian Bubak^{*},
Matei Ripeanu^{*}, Xian-He Sun^{*}, and Hong-Linh Truong^{*}

In recent years, the emergence and evolution of large scale parallel systems, grids, cloud computing environments, and multi-core architectures have prompted the performance community to push its boundaries. Whether system scale is achieved by coupling processors with a large number of cores that are tightly coupled or by massive numbers of loosely coupled processors, many systems will contain hundreds of thousands of processors on which millions of computation threads solve ever larger and more complex problems. At the same time, the coverage of the term 'performance' has constantly broadened to include reliability, robustness, energy consumption, and scalability in addition to classical performance-oriented evaluations of system functionalities. In response to these two new sets of challenges, our community has the mission to develop a range of novel methodologies and tools for performance modeling, evaluation, prediction, measurement, benchmarking, and visualization of existing and emerging large scale parallel and distributed systems.

This topic, "Performance Prediction and Evaluation," brings together system designers and researchers involved with the qualitative and quantitative evaluation and modeling of large scale parallel and distributed applications and systems (e.g., Grids, cloud computing environments, multi-core architectures). To this end, the Program Committee has sought high-quality contributions that cover aspects of techniques, implementations, tools, standardization efforts, and characterization and performance-oriented development of distributed and parallel applications. Especially welcome have been contributions devoted to performance evaluation and prediction of multi-threaded programs, novel and heterogeneous architectures, and web-based systems and applications.

This year we have received 18 papers, from which after a thorough peer-reviewing process (four reviews) we have selected 5 papers for presentation at the conference. We would like to thank all the contributing authors as well as the reviewing team. The accepted papers provide an interesting mix of research directions in the area of performance evaluation and prediction:

- Two papers extend the methodology of performance prediction and evaluation. The paper titled "Characterizing and Understanding the Bandwidth Behavior of Arbitrary Programs on Multi-core Processors" introduces a model based on execution phases that characterizes off-chip memory accesses and bandwidth sharing of arbitrary programs on multi-core processors. The paper titled "A Methodology to Characterize Critical Section Bottlenecks in DSM Multiprocessors" proposes a new methodology for

^{*} Topic Chairs.

determining the critical section bottlenecks of parallel applications, with applications in performance tuning.

- Two papers focus on new tools for performance prediction and evaluation. The paper titled "PSINS: An Open Source Event Tracer and Execution Simulator for MPI Applications" combines two functionalities, trace generation for MPI applications and trace-based simulation, into a performance modeling framework for MPI applications. The paper titled "Hybrid Techniques for Fast Multicore Simulation" presents two hybrid methods for fast and accurate multi-core simulation in order to support the evaluation of hardware design trade-offs.
- Furthermore, the paper titled "Stochastic Analysis of Hierarchical Publish/Subscribe Systems" presents a contribution to modeling and understanding the performance of a hierarchical publish/subscribe system.

Stochastic Analysis of Hierarchical Publish/Subscribe Systems

Gero Mühl¹, Arnd Schröter¹, Helge Parzy jegla¹, Samuel Kounev²,
and Jan Richling¹

¹ Communication and Operating Systems Group, TU Berlin, Germany

² Software Design and Quality Group, University of Karlsruhe (TH), Germany

{g.muehl, arnd.schroeter, parzy jegla, skounev}@acm.org,
richling@cs.tu-berlin.de

Abstract. With the gradual adoption of publish/subscribe systems in mission critical areas, it is essential that systems are subjected to rigorous performance analysis before they are put into production. However, existing approaches to performance modeling and analysis of publish/subscribe systems suffer from many limitations that seriously constrain their practical applicability. In this paper, we present a generalized method for stochastic analysis of publish/subscribe systems employing identity-based hierarchical routing. The method is based on an analytical model that addresses the major limitations of existing work in this area. In particular, it supports arbitrary broker overlay topologies and allows to set workload parameters, e.g., publication rates and subscription lifetimes, individually for each broker. The analysis is illustrated by a running example that helps to gain better understanding of the derived mathematical relationships.

Keywords: Publish/Subscribe, Performance Analysis.

1 Introduction

Publish/subscribe systems were originally motivated by the need for loosely-coupled and asynchronous dissemination of information in distributed event-based applications. With the advent of ambient intelligence and ubiquitous computing, many new applications of publish/subscribe systems have emerged. The main advantage of publish/subscribe is that it makes it possible to decouple communicating parties in time, space, and flow [2].

In a publish/subscribe system, clients can take over the roles of publishers or subscribers depending on whether they act as producers or consumers of information. *Publishers* publish information in the form of *notifications*, while *subscribers* express their interest in specific notifications by issuing subscriptions. Subscriptions are usually defined as a set of constraints on the type and content of notifications and are often referred to as *filters*. A *notification service* delivers published notifications to all subscribers that have issued matching subscriptions. In many cases, the notification service is implemented by a set of *brokers* each

managing a set of *local clients*. The brokers are connected by *overlay links* and a published notification is routed stepwise from the publisher hosting broker over intermediate brokers to all brokers that host subscribers with matching subscriptions. To achieve this, each broker manages a *routing table* that is used to forward incoming notifications to neighbor brokers and local clients. The routing tables are updated according to a *routing algorithm* (e.g., identity-based or covering-based routing [3]) by propagating information about subscriptions in the broker network in form of *control messages*.

With the increasing popularity of publish/subscribe and its gradual adoption in mission critical areas, performance issues are becoming a major concern. To avoid the pitfalls of inadequate Quality of Service (QoS), it is essential that publish/subscribe systems are subjected to rigorous performance analysis before they are put into production. Existing approaches to performance analysis of publish/subscribe systems suffer from some significant limitations. Most research in this area is focused on specific system configurations that are evaluated by means of time- and resource-intensive simulations. Such evaluations are expensive especially when large-scale systems with multiple alternative configurations and workloads have to be considered. We discuss related work in detail in Sect. 5.

In [4], we derived closed form equations for the routing table sizes and the message rate in publish/subscribe systems based on hierarchical identity-based routing. However, this approach required several simplifications and restrictive assumptions seriously limiting its practical value. For example, the broker topology was assumed to be a complete n -ary tree and publishers were only allowed to be connected to leaf brokers. Furthermore, subscriptions were assumed to be equally distributed among filter classes and brokers.

In this paper, we build on previous work and propose a generalized method for stochastic analysis of publish/subscribe systems that apply identity-based hierarchical routing. The method is based on an analytical model that allows to freely set many important system parameters. In particular, arbitrary tree-based broker overlay topologies are supported and publishers as well as subscribers can connect to any broker in the system. Finally, the subscription arrival rates and their lifetimes, as well as the notification publication rates, can be specified for each broker and filter class individually to model locality. The proposed analytical model can be used to predict the routing table sizes as well as the message rate for a given workload and configuration scenario. This allows to evaluate trade-offs across multiple scenarios with minimal overhead. While presenting our method, we consider an exemplary setting that we use as a running example in order to demonstrate the applicability of our approach and obtain some quantitative results that are used to provide insight into the mathematical relationships. In addition, all analytical results presented for the exemplary setting which comprise over 1400 concrete workload and configuration scenarios, have been compared against simulation of the system to confirm the validity of our method.

The remainder of the paper is organized as follows: In Sect. 2, we describe how routing is done in the type of systems we consider. Section 3 introduces the foundational system model and the exemplary setting. Sect. 4 describes our

analysis method in detail. In Sect. 5 related work is reviewed. Finally, the paper is wrapped up in Sect. 6.

2 Publish/Subscribe Routing

Routing in publish/subscribe systems takes place on two levels. On the upper level, the *overlay network*, messages traverse the broker overlay topology based on their content and the currently active subscriptions. On the lower level, messages are routed through the *physical network* between neighboring brokers in the overlay topology. The routing of notifications in the broker overlay network is done according to a publish/subscribe routing algorithm. In this paper, we consider publish/subscribe systems that use *hierarchical routing* [5,3] to route notifications from publishers to subscribers. With this approach, brokers are arranged in a tree-based overlay topology, where one of the brokers is designated as *root broker*. Subscriptions are propagated only upwards in the broker tree from the subscriber hosting brokers towards the root broker and establish routing entries on their way forming the reverse delivery path for matching notifications.

The propagation of a subscription can be suppressed if the delivery of all notifications matching the subscription is already guaranteed by another active subscription propagated previously. With *identity-based* hierarchical routing, which we consider here, this is the case if a subscription matching the same set of notifications has already been forwarded to the parent broker before. Notifications, on the other hand, are always propagated all the way up to the root broker. In addition, notifications are propagated downwards towards subscribers whenever they meet a matching routing entry on their way to the root broker.

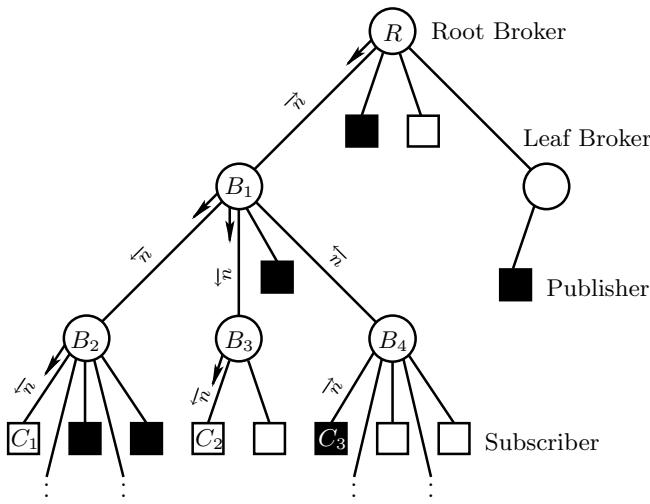


Fig. 1. Hierarchical identity-based routing

The routing algorithm is illustrated in Fig. 1. Client C_1 subscribes to a filter class and the subscription is propagated upwards installing corresponding routing entries in the routing tables of brokers B_2 , B_1 and R (depicted as solid arrows pointing downwards). After that, client C_2 subscribes to the same filter class and the subscription is propagated upwards to B_1 where it meets an already installed routing entry for the respective filter class. Thus, the subscription is not forwarded further up the tree. Then, client C_3 publishes a notification n matching the filter class that the other clients have subscribed to. The notification is propagated from B_4 to its parent broker B_1 which forwards it to the root broker R . Due to the routing entries at B_1 , the notification is additionally forwarded to B_2 and B_3 which deliver it to clients C_1 and C_2 , respectively.

3 Foundational Model and Exemplary Setting

Before we start presenting our analysis method, we first describe the underlying system model as well as an exemplary setting for our running example.

3.1 Foundational System Model

Instead of dealing with clients directly, we assume independent arrivals of new subscriptions at the brokers for each filter class. The subscription inter-arrival times are modeled using exponential distributions. The subscription lifetimes, on the other hand, can have arbitrary distributions. We denote with $\lambda^f(B)^{-1}$ the mean *inter-arrival time* and with $\mu^f(B)^{-1}$ the mean *lifetime* of subscriptions at broker B for filter class f . We denote with $\omega^f(B)$ the *publication rate* of broker B for filter class f . Let \mathcal{B} be the set of all brokers and \mathcal{F} be the set of filter classes. The overall publication rate ω^f of a filter class within the whole system is then $\omega^f = \sum_{B \in \mathcal{B}} \omega^f(B)$ and the overall publication rate over all filters classes is $\omega = \sum_{f \in \mathcal{F}} \omega^f$.

To ease the presentation of our method, we assume that each notification belongs to exactly one filter class and that subscriptions are placed for individual filter classes only. However, the presented approach can be extended from the pure channel-based approach to general identity-based routing. In this case, each subscription belongs to a filter class as before but poses an additional constraint on the notification content. Essentially, this extension effects the probability that two subscriptions are identical and the set of notifications matched by a subscription. Both effects can be addressed by introducing two variable factors depending on the number of subscriptions.

3.2 Exemplary Setting

Next, we present an exemplary setting that we use as a running example in the rest of the paper in order to demonstrate the applicability of our approach and obtain some quantitative results that illustrate the steps of our analysis method and provide insight into the mathematical relationships. To this end, a scenario was chosen that illustrates trade-offs in using different broker topologies and

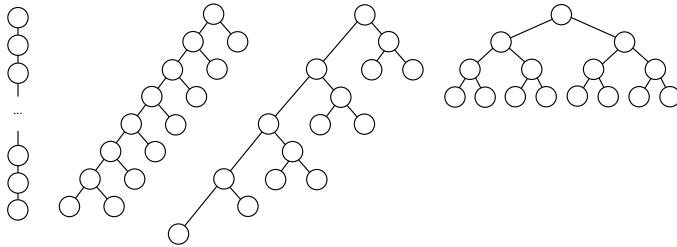


Fig. 2. Resulting topologies for 15 brokers

provides some quantitative results that can be easily interpreted by the reader. Note that all analytical results presented for the exemplary setting (covering in total over 1400 concrete workload and configuration scenarios) have been compared against results obtained through simulation of the system. In all cases, the analytical results were identical to the simulation results.

We consider 7 exemplary topologies consisting of 127 brokers arranged in a binary tree. The trees are varied from a balanced tree with seven levels to a linear arrangement of the brokers with 127 levels by restricting the maximum number of brokers at each level of the tree to 1, 2, 4, 8, 16, 32, 64, respectively. We denote the topologies by T_1 , T_2 , T_4 , T_8 , T_{16} , T_{32} and T_{64} . Fig. 2 shows the four topologies that would result for 15 brokers.

Besides using different topologies, most experiments we present in the context of the example vary the mean overall number of subscriptions \mathcal{N}_s in the system from 10 to 700 000 (equally distributed among brokers and filter classes). Please note that as mentioned in Sect. 1, our analysis method allows arbitrary distributions of the subscriptions and the notifications published among brokers and filter classes. The exemplary setting uses a uniform distribution in order to come up with simple scenarios that illustrate the results. For the rest of this paper, unless stated otherwise, the following parameters are fixed:

- $|\mathcal{F}| = 1000$, where \mathcal{F} is the set of all filter classes.
- Mean lifetime of subscriptions $\mu^f(B)^{-1}$ is 60s for all brokers and filter classes.
- The overall publication rate ω is set to $1000s^{-1}$, i.e., 1000 notifications are published per second (equally distributed among brokers and filter classes).

4 Analysis Method

We now present our analysis method and derive the *routing table sizes* (Sect. 4.1) which are a measure for the matching overhead, as well as the *message rate* (Sect. 4.2) which can be used to estimate the overall communication costs.

4.1 Routing Table Sizes

The routing table of a broker consists of *local routing entries* used to deliver notifications to local subscribers and *remote routing entries* used to forward

notifications to child brokers along delivery paths. The remote routing entries of a broker B depend on the local and remote routing entries of its child brokers. We first introduce $p_0^f(B)$ as the probability that broker B has no *local* subscription for a filter class f . Modeling the subscriptions at the broker with a $M/G/\infty$ queuing system, the probability $p_0^f(B)$ can be determined based on the arrival rate $\lambda^f(B)$ of subscriptions at broker B and their mean lifetime $\mu^f(B)^{-1}$ [6]:

$$p_0^f(B) = e^{-\lambda^f(B)/\mu^f(B)} \quad (1)$$

The expected number of local routing entries for a filter class f is then:

$$x_l^f(B) = \lambda^f(B) \cdot \mu^f(B)^{-1} \quad (2)$$

We now introduce $P_0^f(B)$ as the probability that B has neither a local subscription nor a remote routing entry for filter class f . We denote with $\mathcal{C}(B)$ the set of all child brokers of broker B . $P_0^f(B)$ depends directly on $p_0^f(B)$ and on the value of P_0^f for all child brokers of B (if B is not a leaf broker)¹:

$$P_0^f(B) = p_0^f(B) \cdot \prod_{C \in \mathcal{C}(B)} P_0^f(C) \quad (3)$$

Given that client communication is local, the local routing entries are of minor interest as they do not cause network traffic. We therefore concentrate on the remote routing entries. A broker B has no remote routing entry for a filter class f and a child broker C if C has neither a local subscriber nor a remote routing entry for f . The probability that this is the case is $P_0^f(C)$. The expected number of remote routing entries broker B has for filter class f can then be determined as

$$x_r^f(B) = \sum_{C \in \mathcal{C}(B)} (1 - P_0^f(C)) \quad (4)$$

Thus, the sum of all remote routing entries for filter class f is given by

$$x_r^f = \sum_{B \in \mathcal{B}} x_r^f(B) \quad (5)$$

We denote with $x_r = \sum_{f \in \mathcal{F}} x_r^f$ the expected overall number of remote routing entries in the system. When the number of subscriptions grows infinitely, x_r converges to the product of the number of overlay links and the number of filter classes, i.e., 126 000 in our example.

Figure 3 shows the expected overall number of remote routing entries x_r for the exemplary seven topologies plotted against the mean number of subscriptions in the system. As the number of subscriptions grows, x_r increases strictly monotonically with the gradient² continuously decreasing. As expected, x_r eventually

¹ In case of a leaf broker the (empty) product equals 1.

² Please note that all figures use a logarithmic scale for the number of subscriptions.

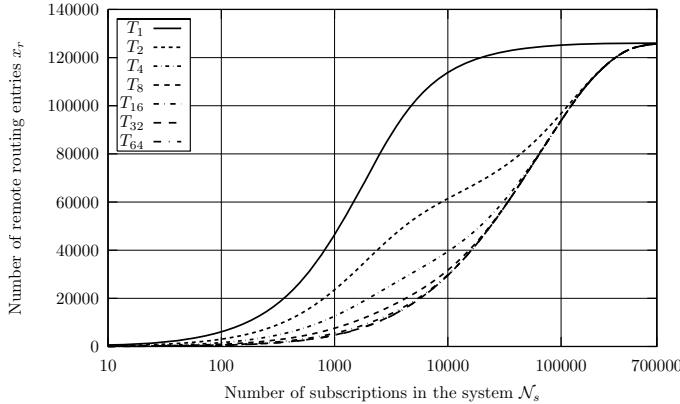


Fig. 3. Routing table sizes

converges to 126 000 for all topologies. The gradient at which the individual plot of a certain topology starts in the origin depends on the average path length of the topology. The longer the average path length, the steeper the plot is for lower number of subscriptions and the flatter it is for higher number of subscriptions.

4.2 Message Rate

The message rate is the sum of the rate used for notification messages and the rate used for control messages.

Notification Rate. In order to derive the notification rate, we first look at the case where all publishers are connected to the root broker. Consider the publication of a notification of filter class f . This notification causes one message for each remote routing entry for filter class f in the system. Thus, the number of notifications b_n sent in the system per second is:

$$b_n = \sum_{f \in \mathcal{F}} x_r^f \cdot \omega^f \quad (6)$$

If publishers can connect to arbitrary brokers, we have to add the number of additional messages sent due to hierarchical routing. A notification forwarded by broker B to its parent broker B' is *additional* if no subscriber for the respective filter class exists in the sub-tree rooted at B since the notification would not have been forwarded over this link from B' to B if it would have been published at the root broker R instead. We will use the notation \hat{B} to denote the parent broker of broker B assuming that $B \neq R$. We introduce $\mathcal{P}(B)$ as the set of brokers on the path from B to R , including B and excluding R , i.e., $\mathcal{P}(B) = \{B\} \cup \mathcal{P}(\hat{B})$ if $B \neq R$ and $\mathcal{P}(B) = \emptyset$ otherwise. The mean number of additional messages caused by a notification of filter class f that is published at a broker B is $\sum_{B' \in \mathcal{P}(B)} P_0^f(B')$ since $P_0^f(B')$ is the probability that there is

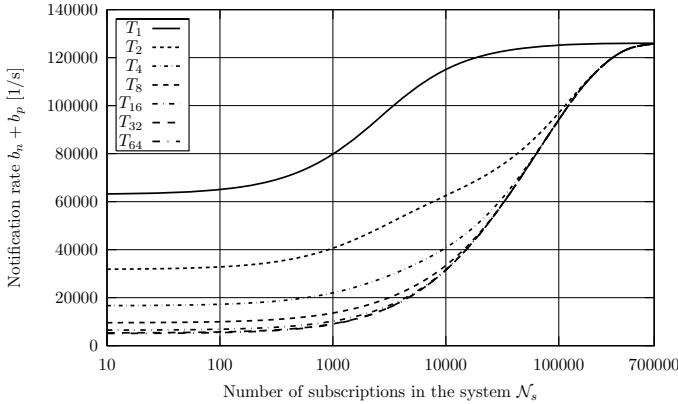


Fig. 4. Overall notification rate

no subscription for filter class f in the sub-tree rooted at B' . Thus, the mean number of additional messages published per second is:

$$b_p = \sum_{f \in \mathcal{F}} \sum_{B \in \mathcal{B} \setminus \{R\}} \left(\omega^f(B) \cdot \sum_{B' \in \mathcal{P}(B)} P_0^f(B') \right) \quad (7)$$

Figure 4 shows the overall notification rate $b_n + b_p$ for our exemplary setting. The rate monotonically increases until it converges to 126 000 notifications per second. For smaller numbers of subscriptions, the rate is dominated by b_p , while for larger numbers of subscriptions it is dominated by b_n . Additionally, the more balanced the topology is, the less important b_p is.

Control Message Rate. The control message rate b_c consists of all messages sent in the system to keep the broker routing tables up-to-date. The key to determine this rate is the toggling of brokers: We say that a broker B is in state 0 for filter class f if it has neither a local nor a remote subscription (i.e., a routing entry installed for one of its child brokers) for this filter class. Otherwise, the broker is said to be in state 1 for this filter class. Since we are using hierarchical identity-based routing (cf. Sect. 2), each time a broker toggles from state 0 to state 1 or vice versa it sends a *toggle message* to its parent broker. The control message rate is then given by the sum of the toggle rates of all brokers, where the toggle rate of an individual broker depends on its local clients and on the toggle rates of its child brokers.

To derive the toggle rate of an inner broker for a given filter class, we need to determine the rate at which subscriptions for the respective filter class arrive in the sub-tree rooted at the broker. We will refer to this rate as the *accumulated* subscription arrival rate. The latter depends on the local subscription arrival rate at the broker, given by $\lambda^f(B)$, and the accumulated arrival rates of its child brokers. The subscription arrivals at each broker form a Poisson process and the

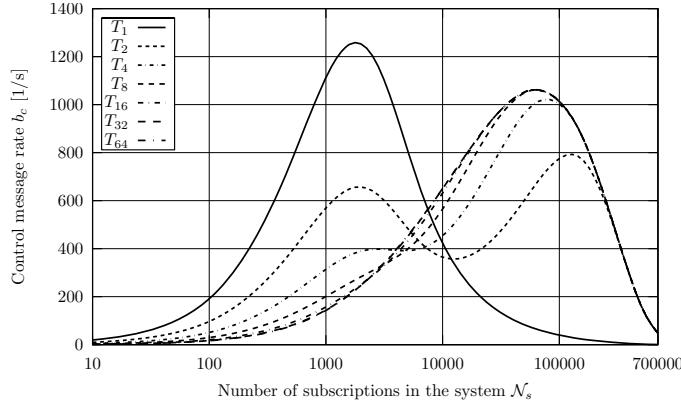


Fig. 5. Control message rate

superposition of multiple independent Poisson processes is also a Poisson process with arrival rate equal to the sum of the arrival rates of the individual Poisson processes. Thus, we obtain the following recursive formula for the accumulated subscription arrival rate $\lambda_a^f(B)$ of broker B for filter class f :

$$\lambda_a^f(B) = \lambda^f(B) + \sum_{C \in \mathcal{C}(B)} \lambda_a^f(C) \quad (8)$$

Using $\lambda_a^f(B)$, we can now calculate the toggle rate $M^f(B)$ of each broker in the system for filter class f . Each subscription issued or revoked leads to a control message sent by broker B to its parent broker if there is no other subscription for the same filter class in the sub-tree rooted in B . The probability for this is $P_0^f(B)$. Moreover, the expected values of the accumulated subscription arrival rate and the accumulated subscription death rate are equal because we consider a system in equilibrium. The expected number of toggles per second is thus:

$$M^f(B) = 2 \cdot \lambda_a^f(B) \cdot P_0^f(B) \quad (9)$$

The control message rate (total number of toggle messages) b_c is given by:

$$b_c = \sum_{f \in \mathcal{F}} \sum_{B \in \mathcal{B} \setminus R} M^f(B) \quad (10)$$

Figure 5 shows the control message rate b_c in the publish/subscribe system for the seven exemplary topologies. For T_1, T_8, T_{16}, T_{32} and T_{64} , the control message rate rises from 0 to a global maximum, then starts to drop, and finally converges to 0 as the number of subscriptions is further increased. This is because for small numbers of subscriptions, the routing tables are only lightly filled and therefore when a subscription is issued or revoked, the probability that some toggle messages are generated is high. However, as the routing tables get increasingly filled, a point is reached after which the probability of generating a

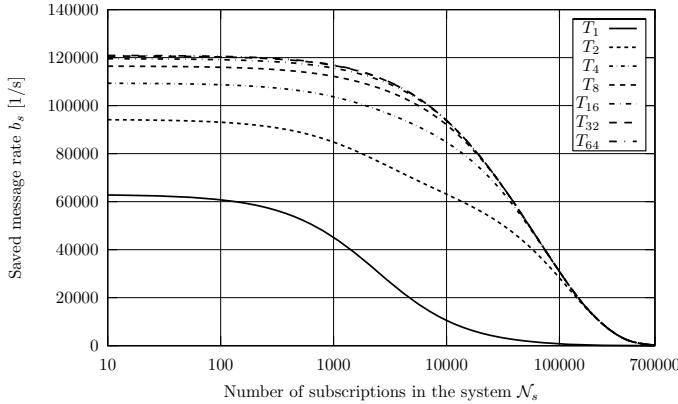


Fig. 6. Saved message rate compared to flooding

toggle message when a subscription is issued or revoked starts to drop. In consequence, the control message rate starts to decrease after reaching its maximum and eventually converges to 0.

For T_1 , the control message rate reaches its maximum for a much smaller number of subscriptions than in the other topologies. This is due to the fact that in this topology the routing tables fill up more quickly. The plots of T_2 and T_4 are different in that they additionally have a local maximum caused by the asymmetry of these topologies.

Comparison with Notification Flooding and Simple Routing. The analysis method presented so far can already be used to compare hierarchical routing to basic notification flooding. With flooding no remote routing entries are used: Each published notification is sent once over every overlay link leading to a notification rate of

$$b_f = \left(\sum_{B \in \mathcal{B}} |\mathcal{C}(B)| \right) \cdot \sum_{f \in \mathcal{F}} \omega^f \quad (11)$$

The notification rate saved by hierarchical routing compared to basic notification flooding is thus given by:

$$b_s = b_f - (b_n + b_p) \quad (12)$$

The overall message rate b saved by applying hierarchical routing compared to using basic notification flooding is then:

$$b = b_s - b_c \quad (13)$$

Figure 6 shows the saved message rate for the seven exemplary topologies. The saved message rate equals 126 000 notifications per second if there are no subscriptions in the system. For larger numbers of subscriptions, the message rate

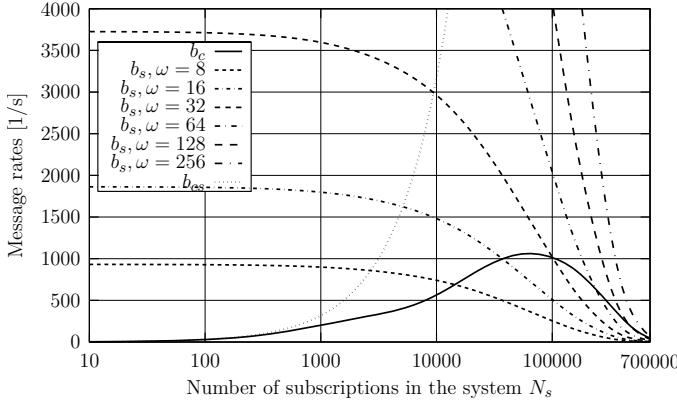


Fig. 7. Control message vs. notification rate (for T_8)

saved continuously decreases and finally converges to 0. The more unbalanced the topology is, the steeper the drop in the saved message rate is.

For our last example, we include a comparison with simple routing, where each subscription is always propagated towards the root broker [3]. In this case, the notification traffic is the same while the control traffic is:

$$b_{cs} = 2 \cdot \sum_{f \in \mathcal{F}} \sum_{B \in \mathcal{B} \setminus \{R\}} \lambda_a^f(B) \quad (14)$$

Figure 7 depicts the control message rates b_c and b_{cs} as well as the saved notification rate b_s for topology T_8 and different publication rates ω . It shows that depending on ω , filtering may perform worse than flooding if the number of subscriptions in the system exceeds a certain threshold. For example, with $\omega = 32 s^{-1}$ this is the case for identity-based routing if $N_s > 100,000$ and for simple routing if $N_s > 10,000$. Identity-based routing outperforms simple routing.

5 Related Work

An analytical model of publish/subscribe systems based on subscription forwarding is presented by Castelli et al. [7]. The authors provide closed form analytical expressions for the overall network traffic required to disseminate subscriptions and propagate notifications, as well as for the message forwarding load on individual system nodes. However, the same restrictive assumptions as in [4] are made about the topology and the distribution of publishers and subscribers among brokers. Thus, this model is not applicable in most practical scenarios.

Bricconi et al. present in [8] a simple model of the JEDI publish/subscribe system. The model is mainly used to calculate the number of notifications received by each broker using a uniform distribution of subscriptions. To model the multicast communication, the authors introduce a spreading coefficient between 0

and 1 which models the probability that a broker at a given distance (in hops) from the publishing broker receives a published notification.

Baldoni et al. [9][10] propose an analytical model of distributed computation based on a publish/subscribe system. The system is abstracted through two delays (subscription/unsubscription delay and diffusion delay) which are assumed to be known. The proposed model is only used to calculate the number of notifications that are missed by subscribers due to high network delays.

A basic high-level cost model of publish/subscribe systems in mobile Grid environments is presented in [11]. This model, however, does not provide much insight into the behavior of the system since it is based on the assumption that the publish/subscribe cost and time delay per notification are known. In [12], probabilistic model checking techniques and stochastic models are used to analyze publish/subscribe systems. The communication infrastructure (i.e., the transmission channels and the publish/subscribe middleware) are modeled by means of probabilistic timed automata. The analysis considers the probability of message loss, the average time taken to complete a task and the optimal message buffer sizes. However, a centralized architecture is assumed.

In [13], a methodology for workload characterization and performance modeling of distributed event-based systems is presented. A workload model of a generic system is developed and analytical analysis techniques are used to characterize the system traffic and to estimate the mean notification delivery latency. For more accurate performance prediction queuing Petri net models are used. While this technique is applicable to a wide range of systems, it relies on monitoring data obtained from the system and it is therefore only applicable if the system is available for testing. Furthermore, for systems of realistic size and complexity, the queuing Petri net models would not be analytically tractable.

6 Conclusions

In this paper, we presented a generalized method for stochastic analysis of publish/subscribe systems employing identity-based hierarchical routing. The method eliminates the major limitations of existing work in this area and is applicable to a much wider range of systems. We presented results for over 1400 different workload and configuration scenarios for which we have compared the analytical results against simulation confirming the validity of our method.

The analysis method we presented provides an important foundation for performance modeling and evaluation of publish/subscribe systems which is an important contribution in an area where evaluations mostly rely on extensive simulation studies and often neither the simulation code nor the underlying data sets are publicly available.

As a next step, we intend to build on our analysis method and derive performance metrics such as utilization and delay for physical links and overlay links as well as notification and subscription delays. Furthermore, we intend to extend our analytical model to support covering-based and merging-based routing [3], peer-to-peer routing in addition to hierarchical routing, and advertisements.

References

1. Oki, B., Puegl, M., Siegel, A., Skeen, D.: The information bus: an architecture for extensible distributed systems. In: SOSP 1993: Proceedings of the fourteenth ACM symposium on Operating systems principles, pp. 58–68. ACM, New York (1993)
2. Eugster, P., Felber, P., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM Computing Surveys 35(2), 114–131 (2003)
3. Mühl, G., Fiege, L., Pietzuch, P.: Distributed Event-Based Systems. Springer, Heidelberg (2006)
4. Jaeger, M.A., Mühl, G.: Stochastic analysis and comparison of self-stabilizing routing algorithms for publish/subscribe systems. In: Proc. of the 13th IEEE/ACM Intl. Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, pp. 471–479 (2005)
5. Cugola, G., Di Nitto, E., Fuggetta, A.: The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. IEEE Transactions on Software Engineering 27(9), 827–850 (2001)
6. Jain, R.: The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling. Wiley Interscience, New York (1991)
7. Castelli, S., Costa, P., Picco, G.P.: Modeling the communication costs of content-based routing: The case of subscription forwarding. In: Proc. of the Inaugural Conference on Distributed Event-Based Systems (DEBS 2007), pp. 38–49 (2007)
8. Bricconi, G., Nitto, E.D., Tracanella, E.: Issues in analyzing the behavior of event dispatching systems. In: Proc. of 10th Intl. Workshop on Software Specification and Design, pp. 95–103 (2000)
9. Baldoni, R., Beraldì, R., Piergiovanni, S.T., Virgillito, A.: Measuring notification loss in publish/subscribe communication systems. In: Proc. of 10th IEEE Pacific Rim International Symposium on Dependable Computing, pp. 84–93 (2004)
10. Baldoni, R., Beraldì, R., Piergiovanni, S.T., Virgillito, A.: On the modelling of publish/subscribe communication systems. Concur. and Comput.: Pract. and Exper. 17(12), 1471–1495 (2005)
11. Oh, S., Pallickara, S.L., Ko, S., Kim, J.-H., Fox, G.C.: Cost model and adaptive scheme for publish/Subscribe systems on mobile grid environments. In: Sunderam, V.S., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2005. LNCS, vol. 3516, pp. 275–278. Springer, Heidelberg (2005)
12. He, F., Baresi, L., Ghezzi, C., Spoletini, P.: Formal analysis of publish-subscribe systems by probabilistic timed automata. In: Derrick, J., Vain, J. (eds.) FORTE 2007. LNCS, vol. 4574, pp. 247–262. Springer, Heidelberg (2007)
13. Kounev, S., Sachs, K., Bacon, J., Buchmann, A.: A methodology for performance modeling of distributed event-based systems. In: Proc. of the 11th IEEE Intl. Symposium on Object/Component/Service-oriented Real-time Distributed Computing (2008)

Characterizing and Understanding the Bandwidth Behavior of Workloads on Multi-core Processors

Guoping Long, Dongrui Fan, and Junchao Zhang

Key Laboratory of Computer Systems and Architecture and
Institute of Computing Technology and Chinese Academy of Science
100080 Beijing and China
{longguoping, fandr, jcchang}@ict.ac.cn

Abstract. An important issue of current multi-core processors is the off-chip bandwidth sharing. Sharing is helpful to improve resource utilization and but more importantly and it may cause performance degradation due to contention. However and there is not enough research work on characterizing the workloads from bandwidth perspective. Moreover and the understanding of the impact of the bandwidth constraint on performance is still limited. In this paper and we propose the phase execution model and and evaluate the arithmetic to memory ratio (AMR) of each phase to characterize the bandwidth requirements of arbitrary programs. We apply the model to a set of SPEC benchmark programs and obtain two results. First and we propose a new taxonomy of workloads based on their bandwidth requirements. Second and we find that prefetching techniques are useful to improve system throughput of multi-core processors only when there is enough spare memory bandwidth.

Keywords: multi-core architecture, phase model, memory bandwidth.

1 Introduction

In recent years and multi-core architectures are explored extensively by chip architects to both exploit parallelism and better meet the power consumption envelope. Commercial processors with two to four cores are prevalent in industry. Many core designs have also been proposed and evaluated.

An important problem of current multi-core architectures is the off-chip memory bandwidth sharing. While bandwidth sharing improves resource utilization sometimes and it can also cause performance degradation of the running program. Moreover and due to the limitation of the pin count and the gap between the required memory bandwidth and which is proportional with the growth rate of the on-chip computation density and and the realistic off-chip memory bandwidth will become larger and larger as more and more cores are to be integrated on a single chip. However and to the best of our knowledge and there is not enough research work on characterizing the workloads from bandwidth perspective. Moreover and the understanding of the impact of the bandwidth constraint on performance is still limited.

In this work and we attempt to provide insight for two questions. One is how to characterize the bandwidth requirements of arbitrary programs. The other one is how

to understand the impact of the shared memory bandwidth constraint on system performance. The answers to both questions are important for both architects and system software developers. First and chip architects need to know the bandwidth characteristics of the application in order to determine the amount of off-chip bandwidth and the appropriate area ratio between the computation logic and on chip storage. Second and understanding bandwidth requirements of applications is also helpful for operating system developers to implement an optimum task scheduler and which can schedule tasks to achieve minimum bandwidth contention.

In this paper and we extend the concept of arithmetic to memory ratio (AMR) to arbitrary programs and propose the phase execution model to characterize the bandwidth behavior of workloads. We partition the program execution into a series of phases and evaluate the AMR of each phase to study the bandwidth requirements. We make three contributions in this paper:

First and we propose an analysis methodology to characterize the bandwidth behavior of workloads. Second and we propose a new taxonomy of workloads based on their bandwidth requirements. Third and we find that prefetching techniques are helpful to improve overall system throughput of multi-core processors only if there is enough spare memory bandwidth.

The rest of the paper is organized as follows. Section 2 presents the phase execution model. Section 3 applies the model to SPEC benchmark programs and discusses our findings. Section 4 discusses related research and the last section concludes the paper.

2 The Phase Model

2.1 The Abstract Processor Model

Before the phase model is defined, we assume an abstract multi-core processor model in order to make our problem scope clear. For each processing core, we model the on-chip memory hierarchy as a fully associative working set memory (WM), with the capacity of C data blocks. There are n processing cores connected together to share the same memory request FIFO queue (MRQ). The on chip inter-connection network could be in any reasonable topology (MESH, bus, crossbar, etc). It includes necessary arbitration logic to deal with situations when multiple cores need to issue requests to MRQ.

2.2 Definition of the Phase Model

We base the phase execution model on the memory trace [1], which contains all information required for understanding the memory system behavior, including the bandwidth requirements. Intuitively, we define a phase as a segment of the memory trace which accesses distinct data blocks. Starting from this simple intuition, we partition the whole memory trace of a program execution into a series of phases and evaluate the AMR of each phase. First, we establish the memory trace concept formally.

Definition 1. *An address element is a 3-tuple $< T_s, T_e, \text{addr} >$, where T_s , T_e and addr denote the time the request is issued, the time the request is acknowledged and the address (aligned to memory block boundary) of the request, respectively.*

We do not distinguish operation type, such as load or store, in the address element. But we do care about whether an address hits in WM or not. If the *addr* field of an address element hits in WM, then $T_s = T_e$. Otherwise an off-chip memory request is generated for *addr*, and $(T_e - T_s)$ denotes the off-chip memory latency of the request. In this paper, we only take the off-chip latency into account, because only off-chip memory requests contribute to the memory bandwidth consumption.

Definition 2. A memory trace is a ordered set A of N address elements: $A = \{A_1, A_2, A_3, \dots, A_N\}$.

We do not model multiple outstanding memory requests simultaneously in this paper. In other words, each element must issue after its previous one has finished. Therefore, $\forall A_k (1 \leq k \leq N)$, we have $A_{k+1}[T_s] > A_k[T_e]$. Based on the memory trace concept, we can formally define the phase concept as follows.

Definition 3. An ordered partition P of a memory trace $A = \{A_1, A_2, A_3, \dots, A_N\}$ is an ordered set of $m + 1 (m \geq 0)$ ordered sets $P = \{P_1, P_2, P_3, \dots, P_{m+1}\}$, where $1 \leq k_1 < k_2 < \dots < k_m \leq N$, $P_1 = \{A_1, A_2, \dots, A_{k_1}\}$, $P_2 = \{A_{k_1+1}, A_{k_1+2}, \dots, A_{k_2}\}$, ..., $P_{m+1} = \{A_{k_m}, A_{k_m+1}, \dots, A_N\}$.

For example, one ordered partition of the memory trace $A = \{A_1, A_2, A_3, A_4, A_5\}$ is: $P = \{\{A_1, A_2\}, \{A_3\}, \{A_4, A_5\}\}$.

Definition 4. The reference set of an ordered set of K address elements $S = \{A_1, A_2, \dots, A_K\}$ is a set of memory addresses $F(S)$ and is defined by the following formula: $F(S) = \bigcup_{i=1}^K \{A_i[\text{addr}]\}$.

Note that the reference set of an ordered set A is not an ordered set. We can now define the notion of the phase trace and execution phases precisely.

Definition 5. A phase trace is an ordered partition $PT = \{P_1, P_2, \dots, P_m\}$ of a memory trace $A = \{A_1, A_2, A_3, \dots, A_N\}$, with the following two conditions both satisfied: (1) $|F(P_1)| = C (C > 0)$; (2) $\forall k (1 < k \leq m), |F(P_k - P_{k-1})| = C$

Definition 6. Given a phase trace $PT = \{P_1, P_2, \dots, P_m\}$, each sub-partition $P_i (1 \leq i \leq m)$ is called an execution phase of the phase trace.

Given an execution phase, we are interested in the ratio between the arithmetic computation time and the off-chip memory latency within the phase. As will be seen in the next section, it has strong impact on the performance when multiple cores shared the memory bandwidth. Precisely, we define arithmetic to memory ratio (AMR) as follows:

Definition 7. The arithmetic to memory ratio (AMR) of a phase $PH = \{E_1, E_2, \dots, E_p\}$ is defined by the following formula:

¹ In this paper, we use [] operator to reference an address element's fields.

² Let A and B be two sets. In this paper, $|A|$ denotes the cardinality of set A . And $(A - B)$ denotes the set of all elements which belong to A but not B .

$$AMR_{PH} = \frac{E_p[T_e] - E_1[T_s] - \sum_{i=1}^p (E_i[T_e] - E_i[T_s])}{\sum_{i=1}^p (E_i[T_e] - E_i[T_s])} \quad (1)$$

Empirically, the larger the WM capacity is, the less the demand with the memory bandwidth. For the same memory trace, the phase trace is dependant on the choice of WM size. The larger WM is, the shorter the phase trace. The phase trace notion captures the intuition which exists in real processors. With the phase trace, we are particularly interested in the AMR of each phase, which reveals runtime program locality and bandwidth requirements.

2.3 Bandwidth Sharing Results

In this section, we focus on a special class of programs, for which all phases of the execution have the same AMR. Now supposing processing cores each executing an instance of such a program, we obtain two interesting results regarding bandwidth sharing. The first result targets such a question: Under what condition the shared bandwidth will be a performance bottleneck for multi-core processors? The second result targets another problem: Although it is well known that prefetching can help to improve performance, but can it always help to improve the overall system throughput on multi-core processors?

While quantitatively studying both problems for arbitrary programs on real systems is prohibitively complex, we give a precise result for a simplified version of the problem based on the phase model. Although simplified, we believe these results have interesting implications and can help to promote understanding of the bandwidth constraint of commercial multi-core processors on real workloads.

Theorem 1. *Suppose n cores share the MRQ and each core executes an instance of the program with uniform arithmetic to memory ratio: PAMR. Then iff $n \leq \lfloor PAMR \rfloor + 1$, there exists a memory request schedule in MRQ that no core suffers from performance loss due to bandwidth sharing.*

The proof is simple and thus is omitted due to page constraint. The main idea is that the data fetching time of each phase can be overlapped with the phase execution time of other phases. A possible schedule is shown in Figure 11(left), in which PF and PE denote the data fetching and execution of the phase, respectively. Theorem 1 demonstrates that the bandwidth requirement is closely related to the AMR of execution phases.

Commercial multi-core processors usually have prefetching capability, which also consumes the shared bandwidth. Based on the phase model, Theorem 2 gives the quantitative relationship between the AMR and overall system throughput on multi-core processors.

Theorem 2. *Suppose n processing cores with prefetching capability share the MRQ and each core executes an instance of the program with uniform arithmetic to memory ratio: PAMR. Then phase prefetching can help to improve overall system throughput iff $n \leq \lfloor PAMR \rfloor$.*

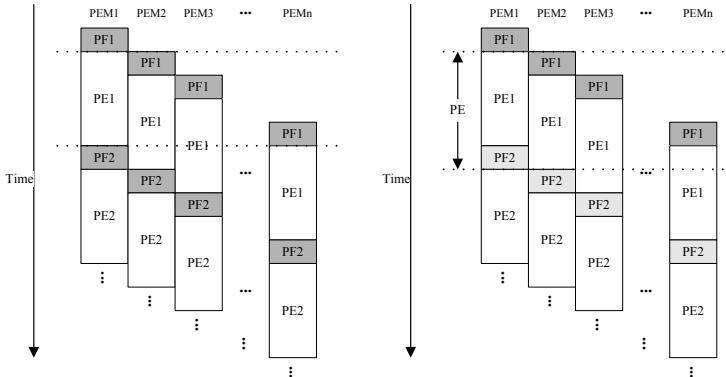


Fig. 1. A schedule without prefetching (left) or with prefetching(right)

We omit the proof here as well due to page constraint. When we say a core has prefetching capability, we mean it can prefetch data for the next phase during the execution of the current phase, and we assume the prefetch buffer is large enough to hold all prefetched data. Figure 1 (right) shows the schedule with prefetching which illustrates the theorem. The most important implication of Theorem 2 is that, prefetching can help to improve overall system throughput only if there is enough spare memory bandwidth. Note that although in uniprocessor systems there is usually spare bandwidth for data prefetching, however, this is not always true for multi-core processors in which all cores share the bandwidth.

3 Application of the Phase Model

3.1 Experimental Methodology

In this section, we apply the phase model to SPEC benchmark programs. Since the model is based on the memory trace, we need a processor pipeline model to generate accurate traces. The simulation environment which we use to collect traces is the Godson-T simulator [234], which is designed for many-core simulation. The processing core has an in order 9-stage instruction pipeline and implements MIPS instruction set. Since the memory trace strongly depends on the processor architecture, please refer to [34] for more description of the details. The benchmark programs are given in Table 1. We run each program for ten billion instructions. All memory traces are analyzed assuming the WM size of 1MB. We conduct two sets of analysis based on the memory trace obtained from the simulator. The first one is the bandwidth behavior analysis, which is done by calculating the AMR distribution of each program. The other one is the performance impact analysis of shared bandwidth. We develop a phase analysis tool to predict the performance loss of each program. The inputs to the tool are phase traces generated from the memory traces of different programs. And the outputs are performance degradation results for all programs due to bandwidth sharing. The shared bandwidth is modeled by a FIFO queue. The MRQ can accept one phase fetch request a time from one phase trace. All requests in MRQ are processed in strict FIFO order.

Table 1. Benchmark Classification

Bandwidth requirements	Program	# phases	AVG. AMR
Low	164.gzip	732	102.8
	458.sjeng	927	103.2
	464.h264ref	136	433.1
Medium	188.ammp	1393	44.4
	401.bzip2	1931	24.2
	175.vpr	5304	23.8
	300.twolf	6657	22.9
High	179.art	66389	0.67
	183.equake	9910	5.74
	470.lbm	64911	1.37
	433.milc	18443	2.37
	482.sphinx3	19183	4.12
	181.mcf	8438	3.11

In this paper, a memory block refers to the data transfer unit between the processor and memory. In cache based systems, a memory block is a last level cache line. In some processors [5], the front side bus supports burst data transfer of multiple words within a memory block. The analysis tool abstracts all implementation details of memory system away and model it as a bandwidth parameter B , which denotes the latency required to bring a memory block from off-chip memory. Suppose a phase fetch request needs to bring C blocks of data to working set memory, the latency introduced by a phase fetch request without bandwidth contention is $B * C$.

3.2 Bandwidth Behavior Analysis

The third and fourth columns of Table 1 show the number of phases observed and the average AMR of each program, respectively. Based on average AMR, we can classify all programs into three categories. Six programs (179.art, 183.equake, 181.mcf, 470.lbm, 433.milc and 482.sphinx3) have very low average AMR, indicating high off-chip bandwidth requirements. Three programs (164.gzip, 458.sjeng and 464.h264ref) have relatively small number of phases and very high average AMR, indicating very low bandwidth requirements. There are also programs (188.ammp, 401.bzip2, 175.vpr and 300.twolf) with medium average AMR, indicating medium demand of bandwidth.

Figure 2(a) to (f) show the AMR distribution for six programs. The x-axis denotes execution time, which is measured as the number of phases executed. The y-axis denotes the AMR of each phase. For the other seven programs, 300.twolf, 179.art, 470.lbm, 181.mcf, 433.milc and 482.sphinx have similar histograms with 175.vpr, and 183.equake has similar histogram with 401.bzip2. We omit the AMR histograms of these programs due to page constraint. As can be seen, different programs have dramatically different AMR distributions. Even for the same program (188.ammp, 164.gzip, 458.sjeng, etc), AMR distributions at different execution periods are also different. We can also divide all thirteen programs into three categories according to the AMR distribution: (1) Programs with periodic execution behavior, including 164.gzip, 188.ammp, 401.bzip2 and 458.sjeng. (2)

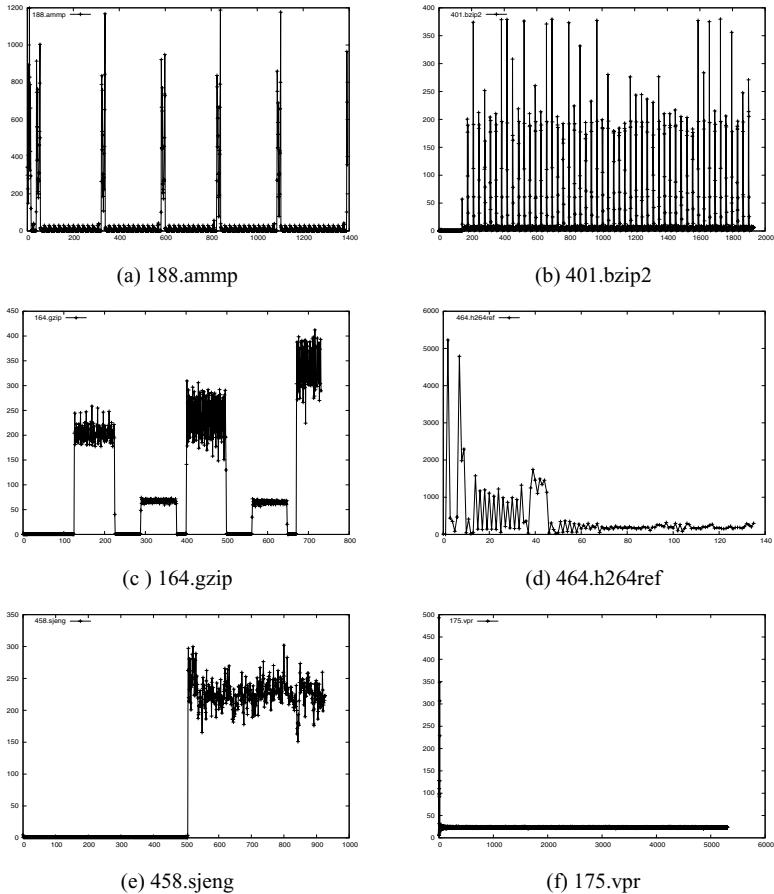


Fig. 2. AMR distribution of program phases

Programs with uniform behavior, including 175.vpr and all other 7 programs which are not shown. (3) Programs with irregular behavior, including 464.h264ref.

3.3 Bandwidth Sharing of Multiple Program Instances

Figure 3 shows results for multiple instances of the same program sharing the off chip memory bandwidth, assuming all instances are started at the same time. The x-axis denotes the number of cores, and the y-axis denotes the normalized performance of the worst core due to bandwidth sharing.

The average AMR has important impact on bandwidth contention. Some programs, including 183.equake, 482.sphinx3, 181.mcf, 433.milc, 470.lbm and 179.art, exhibit poor bandwidth sharing behaviors. This can be partially explained by their low average AMRs (< 6). For programs with relatively higher average AMRs (> 20), including 175.vpr, 300.twolf, 401.bzip2 and 188.ammp, the performance starts to drop when the number of cores increases beyond 16. The results for 464.h264ref, 164.gzip and

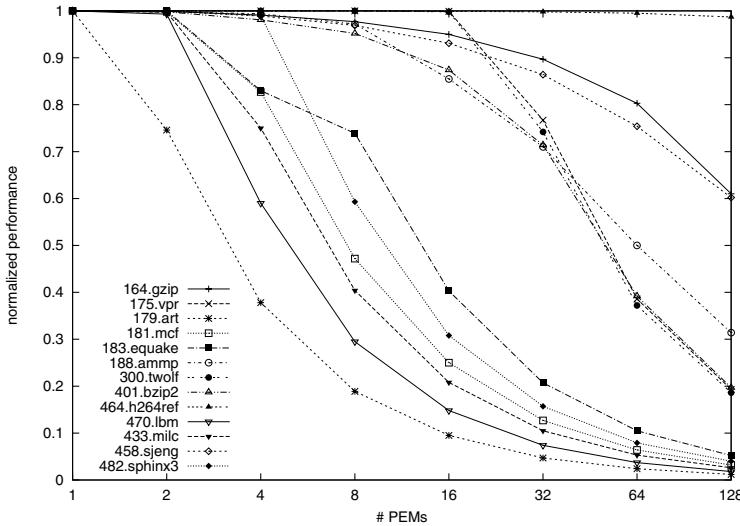


Fig. 3. Bandwidth sharing of multiple program instances

458.sjeng are much better than other programs. Although 164.gzip and 458.sjeng have very large average AMR (> 100), performance gets worse when PEM number is larger than 64 mainly due to the periodic nature of their phase behavior.

For programs (175.vpr and 300.twolf) with uniform phase behaviors, the performance loss due to bandwidth contention only occurs when core number is larger than the average AMR, as demonstrated in Theorem 1. However, for programs (164.gzip and 458.sjeng) with periodic behavior, the contention overhead slows down the performance when core number (64) is much less than the average AMR (> 100).

3.4 Bandwidth Sharing of Multiple Programs

Bandwidth contention of multiple programs is more complex than that of multiple instances of the same program. Table 2 shows normalized performance data (normalized to performance without bandwidth contention) when any two of the programs share the MRQ. For example, if 179.art and 181.mcf run together, the normalized performances of 179.art and 181.mcf are 0.95 and 0.91, respectively. That is, the performance overheads due to bandwidth sharing for 179.art and 181.mcf are 5% and 9%, respectively.

There are two observations to note in Table 2. First, for programs with high average AMR, bandwidth contention is not a problem because only two programs share the bandwidth. Second, if a program P1 with low average AMR and another program P2 with relatively high average AMR run together, the performance loss due to contention for P2 is worse than P1. For example, bandwidth sharing can cause higher performance loss to 181.mcf, 183.eqquake, 300.twolf, 470.lbm, 433.milc, and 482.sphinx3 than to 179.art. Operating systems with runtime AMR information can avoid scheduling programs with poor locality on cores of the same processor.

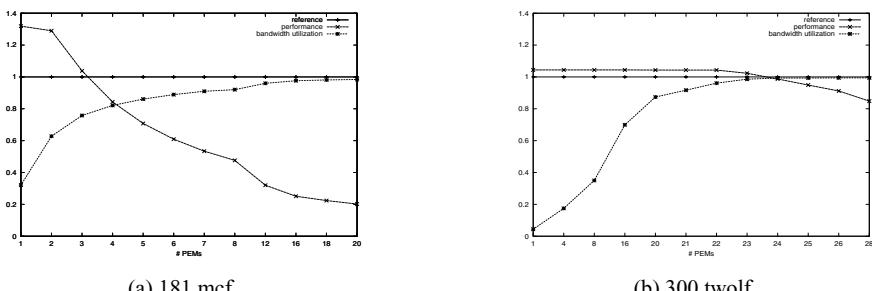
Table 2. Performance Results of Two Programs Sharing the Bandwidth

	gzip	Vpr	Art	Mcf	eqk	Amp	twlf	Bzip	h264	Lbm	milc	seng	sphx
Gzip	1.0	1.0	0.99	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Vpr	1.0	1.0	0.95	0.97	1.0	1.0	1.0	1.0	1.0	0.99	0.98	1.0	0.99
Art	1.0	1.0	0.74	0.95	0.98	0.99	0.99	0.99	1.0	0.91	0.97	1.0	0.96
Mcf	1.0	1.0	0.91	0.99	0.99	0.99	1.0	0.99	1.0	0.99	0.97	1.0	0.99
Eqk	1.0	1.0	0.88	0.94	1.0	1.0	1.0	0.99	1.0	0.96	0.97	1.0	0.98
Amp	1.0	1.0	0.98	0.99	0.99	0.99	1.0	1.0	1.0	0.99	0.99	1.0	0.99
Twlf	1.0	1.0	0.94	0.98	0.99	0.99	1.0	1.0	1.0	0.98	0.99	1.0	0.99
Bzip	1.0	1.0	0.98	0.99	0.99	1.0	1.0	0.99	1.0	0.99	0.99	0.99	0.99
H264	1.0	1.0	0.99	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Lbm	1.0	1.0	0.74	0.94	0.99	1.0	0.99	0.99	1.0	0.99	0.95	0.99	0.98
Milc	1.0	0.99	0.85	0.92	0.98	0.99	0.99	0.99	1.0	0.94	0.99	1.0	0.96
Seng	1.0	1.0	0.99	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
sphx	1.0	1.0	0.87	0.95	0.99	0.99	1.0	0.99	1.0	0.94	0.97	1.0	1.0

3.5 Prefetching Effects

Figure 4 shows results for multiple program instances sharing the bandwidth with phase prefetching. The x-axis denotes the core number. For the reference curve and performance curve shown in the figure, the y-axis denotes the performance normalized to execution without bandwidth contention and phase prefetching. And for the bandwidth utilization curve, the y-axis denotes the utilization of memory bandwidth. Only two representative programs, 181.mcf (Figure 4(a)) and 300.twolf (Figure 4(b)) are shown due to page limits. Note that the methodology can be applied to all other programs.

As can be seen, phase prefetching is a very effective technique to hide memory latency for 181.mcf. Without bandwidth contention, performance improvement reaches more than 30%. However, performance drops quickly as the core number becomes larger than 3, even if there is spare memory bandwidth. For 181.mcf, phase prefetching is helpful only if bandwidth contention is very small. For 300.twolf, phase prefetching can only bring limited performance improvement (4%) because the average AMR is relatively large (22.9). In this case, normalized performance drops below 1 only when



(a) 181.mcf

(b) 300.twolf

Fig. 4. Bandwidth sharing of multiple program instances

Table 3. Phase Model Accuracy

Name	2 Cores	4 Cores	8 Cores	16 Cores	32 Cores	64 Cores
164.gzip	0.20%	0.10%	0.10%	0.00%	0.00%	0.00%
175.vpr	0.00%	0.00%	0.00%	0.00%	4.70%	2.20%
181.mcf	4.40%	3.50%	0.40%	0.10%	0.10%	0.10%
300.twolf	0.00%	0.00%	0.00%	0.10%	2.80%	2.30%
179.art	1.30%	0.00%	0.00%	0.00%	0.00%	0.00%
183.equake	3.60%	2.90%	1.30%	0.20%	0.10%	0.10%
188.ammp	0.10%	0.10%	0.20%	0.20%	0.10%	0.40%
401.bzip2	0.20%	0.20%	0.20%	0.30%	3.30%	2.10%
458.sjeng	0	0.00%	0.00%	0.00%	0.00%	0.00%
464.h264ref	0.10%	0.00%	0.00%	0.00%	0.00%	0.00%
470.lbm	4.50%	3.20%	1.20%	0.60%	0.30%	0.00%
433.milc	4.90%	1.30%	0.10%	0.10%	0.10%	0.10%
482.sphinx3	0.60%	1.70%	2.60%	0.00%	0.00%	0.00%

the core number is larger than 23, for two reasons: (1) The bandwidth usage of 300.twolf saturates ($> 99\%$). As demonstrated in Theorem 2, prefetching is useful only when there is spare memory bandwidth; (2) Bandwidth contention also results in performance loss.

3.6 Phase Model Validation

Previous experimental results are based on the phase analysis tool. To validate the accuracy of the analysis, we compare the predicted results obtained from the tool (shown in Figure 3) against that obtained with cycle accurate execution of these programs on Godson-T many-core simulator. Table 3 shows the results for each program. Start from the second column, each one shows the performance variation when multiple instances of the same program are used as input to the tool. For example, the 8Core column shows the performance variation between the performance predicted by the tool and the cycle accurate execution performance when eight instances are running simultaneously. As can be seen from the table, the maximum performance variation for all programs is within 4.9%. Therefore, the phase model is reasonably accurate for the basic performance trend analysis of bandwidth sharing. Note that although we only present results for multiple instances of the same program, analysis of the sharing behavior of different programs can be done easily as well.

4 Related Work

Bandwidth sharing is a critical issue for multi-core processors. Previous works [6, 7, 8] have been done to understand the bandwidth sharing in multi-core processors. In this work, we take a different approach by generalizing the arithmetic to memory ratio to general applications, and report our results on the impact of bandwidth sharing on performance.

Characterizing programs with arithmetic to memory ratio (AMR) has previously appeared in works on stream or many-core processor designs [9], for which programs with

regular arithmetic structure and high computation density can get high performance. In this paper, we extend the AMR concept to characterize the behavior of arbitrary programs. We show that the AMR distribution of execution phases has important connection with the bandwidth demands.

The idea of execution phase in this paper is partially inspired by previous works on performance optimizations [2][10][11]. [2][10] proposed optimization techniques for IBM C64 architecture, the idea is to partition the program into multiple phases, and overlap the computation and data transfer between the processor and memory explicitly. [11] proposed a new prefetching technique, called epoch model, to improve the processor performance. In this paper, we define the phase model formally above the memory trace. And study the AMR distribution among all execution phases to understand the bandwidth requirements of the program on multi-core architectures.

5 Conclusion

Bandwidth sharing is an important problem for current multi-core processors. In this paper, starting from the memory address trace, we formally define the phase model. The AMR distribution among all phases in the phase trace provides an interesting window to understand the bandwidth behavior of arbitrary programs. Based on the phase trace, we develop a trace driven bandwidth analysis tool to quantitatively study the impact of shared memory bandwidth constraint on performance. More importantly, we propose a new angle to classify the workloads based on bandwidth requirements.

There are two important conclusions from our experimental results. First, the bandwidth requirement of a program depends on the AMR distribution of its phase trace. Programs with high average AMR are more sensitive to bandwidth sharing than those have low average AMR. Second, in multi-core processors with shared bandwidth, prefetching techniques are useful to improve overall throughput only when there is spare memory bandwidth.

Different application areas may have dramatically different demands on off-chip memory bandwidth. For chip architects, the phase model is helpful to understand the bandwidth behavior of applications. In future work, we will investigate the possibility of passing runtime bandwidth information to OS kernels. With this information, the OS kernel can schedule tasks with minimum bandwidth contention to multiple cores of the same processor.

Acknowledgements

This work is supported by the national grand fundamental research 973 program of China under grant No. 2005CB321600, the national high-tech research and development (863) plan of China under grant No. 2009AA01Z103, the national natural science foundation of China under grant No. 60736012, the EU MULTICUBE project under grant No. FP7-216693, and Beijing natural science foundation under grant No. 4092044. In particular, the authors would like to thank anonymous reviewers for their constructive comments to this paper.

References

1. Uhlig, R., Mudge, T.: Trace-driven memory simulation: A suvey. *ACM Computing Surveys* 29(2) (June 1997)
2. Tan, G.M., Fan, D.R., Zhang, J.C., Russo, A., Gao, G.R.: Experience on optimizing irregular computation for memory hierarchy in manycore architecture. In: *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (February 2008)
3. Yuan, N., Yu, L., Fan, D.: An efficient and flexible task management for many-core architecture. In: *Proceedings of Workshop on Software and Hardware Challenges of Manycore Platforms*, In conjunction with the 35th International Symposium on Computer Architecture (June 2008)
4. Long, G.P., Fan, D.R., Zhang, J.C.: Architectural support for cilk computations on many-core architectures. In: *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (February 2009)
5. Hu, W.W., Zhang, F.X., Li, Z.S.: Microarchitecture and performance of godson-2 processor. *Journal of Computer Science and Technology* 20(2) (2005)
6. Rob, A.P., Mandal, F.A., Lim, M.Y.: Empirical evaluation of multi-core memory concurrency initial version (January 2009)
7. Weidendorfer, J.: Understanding memory access bottlenecks on multicore (2007)
8. Ahsan, B., Zahran, M.: Cache performance, system performance, and off-chip bandwidth... pick any two. In: *Proceedings of INA-OCMC* (2009)
9. Long, G.P., Fan, D.R., Zhang, J.C., Song, F.L., Yuan, N., Lin, W.: A performance model of dense matrix operations on many-core architectures. In: *Proceedings of European Conference on Parallel and Distributed Computing* (August 2008)
10. Tan, G.M., Sun, N.H., Gao, G.R.: A parallel dynamic programming algorithm on a multi-core architecture. In: *Proceedings of the Annual ACM Symposium on Parallelism in Algorithms and Architectures* (2007)
11. Chou, Y.: Low-cost epoch-based correlation prefetching for commercial applications. In: *Proceedings of International Symposium on Microarchitecture* (2007)

Hybrid Techniques for Fast Multicore Simulation*

Manu Shantharam, Padma Raghavan, and Mahmut Kandemir

Department of Computer Science & Engineering,
Pennsylvania State University, University Park, PA 16802, USA
{shanthar,raghavan,kandemir}@cse.psu.edu

Abstract. One of the challenges in the design of multicore architectures concerns the fast evaluation of hardware design-tradeoffs using simulation techniques. Simulation tools for multicore architectures tend to have long execution times that grow linearly with the number of cores simulated. In this paper, we present two hybrid techniques for fast and accurate multicore simulation. Our first method, the Monte Carlo Co-Simulation (MCCS) scheme, considers application phases, and within each phase, interleaves a Monte Carlo modeling scheme with a traditional simulator, such as Simics. Our second method, the Curve Fitting Based Simulation (CFBS) scheme, is tailored to evaluate the behavior of applications with multiple iterations, such as scientific applications that have consistent cycles per instruction (CPI) behavior within a subroutine over different iterations. In our CFBS method, we represent the CPI profile of a subroutine as a signature using curve fitting and represent the entire application execution as a set of signatures to predict performance metrics. Our results indicate that MCCS can reduce simulation time by as much as a factor of 2.37, with a speedup of 1.77 on average compared to Simics. We also observe that CFBS can reduce simulation time by as much as a factor of 13.6, with a speedup of 6.24 on average. The observed average relative errors in CPI compared to Simics are 32% for MCCS and significantly lower, at 2%, for CFBS.

1 Introduction

Software simulation models have been used to evaluate the performance of computer hardware for over two decades due to low prototyping costs [3, 5, 7, 10]. With a recent shift in processor design to multicores [1, 2, 17, 20, 21], as a result of technological advances and packaging limits, existing software simulation models for single core processors [3, 16] have been expanded to simulate multicore environments [5, 9, 10, 13, 19]. However, the current generation multicore simulators are slow [1] and the performance degrades as the number of simulated cores increases. This performance degradation is especially visible when large multithreaded benchmarks are used. The execution of these codes on modern multicore simulators is so slow that researchers report experimental results based on partial runs that are not representative of the execution of the entire benchmarks [11, 14]. Hence, there is a need for alternate software based simulation techniques for fast and accurate multicore architecture exploration.

* This work was funded in part through grants 0720749, 0444345, 0811687, 0720645, and 0702519 from the National Science Foundation.

In this paper, we propose two hybrid techniques for fast and accurate multicore simulation. Our first method, called the M CCS (Monte Carlo Co-Simulation), divides application execution into phases with each phase representing a subroutine, or a set of subroutines. Each phase is partially run using Simics [10] and partially using a Monte Carlo predictive model. Our proposed M CCS scheme reduces the simulation time by a factor of 1.77 on average, with average relative error in CPI (cycles per instruction) of 32% compared to Simics. Our second method, CFBS (Curve Fitting Based Simulation), is based on the observation that most scientific applications have multiple iterations and their behavior is repetitive over various iterations. Thus, we represent the CPI profile of a subroutine as a *signature* using curve fitting and represent the entire application execution as a set of signatures to predict performance metrics. Our results indicate that the CFBS scheme reduces the simulation time by a factor of 6.24 on average, with average relative error in CPI of 2% compared to Simics.

We would like to observe that our proposed methods, M CCS and CFBS, are different qualitatively from the previously proposed techniques like Simflex [13] and Simpoint [4] that help reduce the actual simulation time. Section 7 gives a detailed comparison between our methods and these techniques.

The reminder of this paper is organized as follows. In Section 2, we discuss our experimental setup and the tools we use. In Sections 3 and 4, we present the proposed M CCS and CFBS schemes respectively. In Section 5, we present experimental results and in Section 6, we report on a sensitivity study. In Section 7, we describe related work, and in Section 8, we conclude this paper with a summary of our major results.

2 Experimental Setup

We use our schemes, M CCS and CFBS, in conjunction with a full system simulator. For our experiments, Simics [10], a full system simulator, is configured as a multicore

processor to obtain the necessary inputs for our models. For our base system, we configure Simics to simulate a four-core UltraSPARC-III architecture [6] with a 64 KB private level 1 (L1) cache per core, and a 4MB shared level 2 (L2) cache. Table 2 lists the various parameters used in our base configuration and their values. Additionally, in our base configuration, we use a window size of 10 million instructions for the CPI calculation.

Table 1. Benchmark description. NAS: FT, BT, CG; SPLASH2: Barnes, Ocean, Raytrace.

FT	Computational kernel of 3-D FFT-based spectral method. Performs three 1-D FFT, one for each dimension.
BT	Simulates CFD application that uses an implicit algorithm to solve 3-D compressible Navier-Stokes equations.
CG	Conjugate Gradient method to compute an approximation to the smallest eigenvalue of a large, sparse, unstructured matrix.
Barnes	Simulates the interaction of a system of bodies in 3-D over a number of time steps using Barnes-Hut N-body method.
Ocean	Simulates large-scale ocean movements based on eddy and boundary currents.
Raytrace	Renders a 3-D scene onto a 2-D image plane using optimized ray tracing.

We use a subset of NAS OMP [18] and SPLASH2 [12] benchmark suites with varying workload characteristics. In our experiments, we use three NAS parallel OMP benchmarks: FT, BT and CG, each with problem size ‘A’ and three SPLASH2 benchmarks: Barnes, Raytrace and Ocean. A brief description of these benchmarks is included in Table 1. The current benchmark selection is based on the ease of identifying the phases in the code manually.

3 Monte Carlo Co-Simulation (MCCS)

In this section, we discuss MCCS which uses a Monte Carlo predictive model in conjunction with Simics to predict performance metrics of scientific applications. In Section 3.1, we describe our Monte Carlo predictive model. In Section 3.2, we introduce the concept of *windowed CPI* that is useful in understanding the behavior of our target applications and finally, in Section 3.3, we describe the simulation flow of our MCCS scheme.

3.1 Monte Carlo Predictive Model

We model processor cores and the memory subsystem, and consider four basic types of instructions - load, store, floating point and “other”. The “other” instruction type includes all instructions that are not loads, stores or floats. We further classify a *load* instruction as *load L1 hit*, *load L2 hit* or *load memory bound*. Thus, any application execution can be represented as a mix of our six instruction types (floating point, store, load L1 hit, load L2 hit, load memory bound, and “other”). The processor is modeled as a unit that issues these six types of instructions based on their probability of occurrence. The input parameters to our model are the total number of instructions to be issued and the histograms representing the instruction mix. The output of our model is a CPI distribution. Figure 1 shows a pictorial representation of our model for a two-core processor, where p_1 , p_2 , p_3 are the probabilities that a load instruction is satisfied in L1 cache, in L2 cache, or in memory, respectively, and p_0 is the probability that the instruction is a

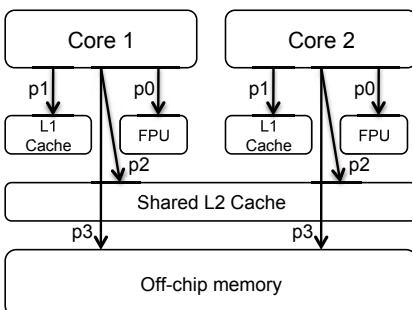


Fig. 1. Monte Carlo predictive model for a two-core processor

Table 2. Parameters used in our base configuration

L1 cache	private, 64KB 2 way assoc
L2 cache	shared, 4MB 16 way assoc
L1 cache latency	3 cycles
L2 cache latency	10 cycles
Off-chip memory latency	260 cycles

floating point instruction¹. We use Simics to obtain the application instruction mix in terms of our six instruction types, and to calculate the probability of their occurrence. Note that Figure 1 is more of an instruction flow diagram based on certain probabilities and it does not represent the actual hardware.

3.2 Understanding Application Performance Using Windowed CPI

Performance is generally measured in terms of average CPI or average IPC of an application. However, these metrics do not provide a detailed insight into application's performance behavior over time, i.e., these metrics do not reveal if the application performance is uniform or variable over time. In order to gain insight into time-dependent performance behavior of applications, we divide its execution time into windows, where a window comprises a fixed number of instructions, for example, 10 million instructions. The CPI for a window is called as *windowed CPI* and unless stated otherwise, CPI in this paper refers to *windowed CPI*.

3.3 MCCS Simulation Flow

In this section, we describe the simulation flow of our MCCS method. We observe that the CPI behavior of an application varies significantly across subroutines and there are intervals of execution within a subroutine, wherein the CPI behavior is uniform. Based on these observations, we divide the application's execution into phases, each phase representing a subroutine or a set of subroutines. A set of subroutines is represented as a phase if the execution time of individual subroutines is very short. At present, we are using an offline phase detection method wherein we manually identify the beginning and the end of subroutines (phases) using Simics *breakpoints*.

We integrate our Monte Carlo predictive model with Simics to take advantage of these phases, and call this integrated method the MCCS. Each application phase is simulated with Simics and Monte Carlo model in an interleaved manner. For each phase, simulation starts with the Simics timing model. After executing a fixed number of instructions, we switch to timing model of our Monte Carlo predictive technique. During the Simics run, we construct histograms representing the probabilities of occurrence of our instruction types for every *window* within a phase. The histograms capture the phase-wise behavior of the application during the Simics execution. We use these histograms in our Monte Carlo predictive model to predict the latencies of our instruction types. While in the Monte Carlo predictive mode, Simics is executed in background without its timing model ("Simics ff"). Because of "Simics ff" mode, Simics does not capture any latencies caused by memory subsystem. We account for these latencies in our Monte Carlo predictive model. We use Simics in "Simics ff" mode to ensure correctness of benchmark execution.

In our experiments, we interleave the executions of Simics and our model twice within a subroutine so that we capture the heterogeneous behavior, i.e., the intervals with uniform CPI behavior within a subroutine. A simple execution flow diagram for this model is shown in Figure 2 where there is a single instance of interleaving of our

¹ Note that since we target scientific applications, we consider only floating-point data. Our strategy can easily be extended to other data types as well.

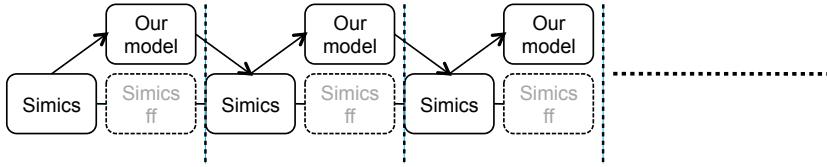


Fig. 2. MCCS simulation flow. The vertical dashed lines represents phase boundaries, “Simics ff” represents Simics’s fast forward execution mode.

model with Simics. The expected accuracy and speed of MCCS is dependent on the number of *interleavings* within a phase. The more the interleavings, the better would be the accuracy and slower would be the simulation. We have fixed the number of interleavings to two per phase so that we could tradeoff between speed and accuracy.

4 Curve Fitting Based Simulation (CFBS)

In this section, we introduce our second hybrid simulation technique, referred to as the CFBS. This technique is tailored to evaluate the behavior of scientific applications on multicores. We observe that scientific applications have multiple iterations of the same code sequences. For example, consider Figure 3 which gives the pseudo code for the NAS FT benchmark. Notice that *fft* subroutine is called repeatedly over different iterations and *fft* subroutine in turn calls subroutines *cffts1*, *cffts2*, *cffts3*.

```

subroutine fft(dir, x1, x2)
.
.
.
do iter = 1, niter
    call evolve(u0, u1, ...)
    call fft(-1, u1, u1)
    call checksum(iter, u1, ...)
end do
.
.
.
end

```

Fig. 3. Left: Code segment of FT benchmark showing *fft* subroutine being called multiple times within a loop. **Right:** Code segment of FT benchmark showing *fft* subroutine.

A detailed analysis of application behavior shows that the CPI behavior of our target applications is repetitive over their entire execution. Consider as an example Figure 4, which shows the CPI behavior of all iterations of the *cffts3* subroutine of NAS FT benchmark. Observe that the CPI behavior of *cffts3* subroutine over different iterations exhibit similar CPI behavior. This repetitive behavior is due to the same memory access pattern and similar cache behavior of this subroutine over different iterations. We use this property of subroutines to reduce the simulation time of scientific applications.

In the CFBS scheme, we first run Simics for one iteration of the application. During this run, we compute the windowed CPIs for each subroutine call and use a *curve fitting method* to fit these data points (windowed CPIs) per subroutine. There are different

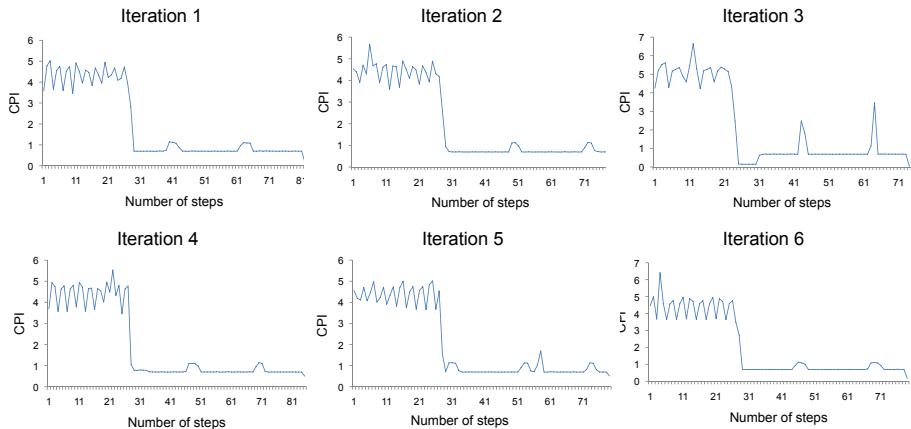


Fig. 4. CPI behavior of cffts3 function for all six iterations. We see that all iterations exhibit a similar behavior.

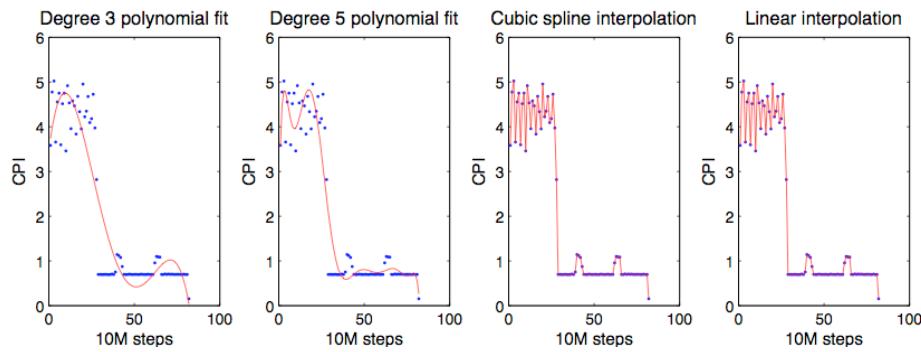


Fig. 5. Curve fitting techniques

```

do iter = 1, 5
  call subroutine 1
  call subroutine 2
  call subroutine 3
end do

```

➡

```

call subroutine 1
call subroutine 2
call subroutine 3
do iter = 2, 5
  subroutine 1 signature
  subroutine 2 signature
  subroutine 3 signature
end do

```

Fig. 6. Left: Actual code representation. **Right:** Representation in CFBS model.

ways in which we can fit a curve to the given data [15]. Figure 5 illustrates four curve fitting techniques, a curve fit with polynomial of degree 3 ($0.0004X^3 - 0.0212X^2 + 0.3057X + 3.4483$), a curve fit with polynomial of degree 5 ($0.0002X^5 - 0.006X^4 + 0.0883X^3 - 0.6928X^2 + 2.4728X + 1.7$), a cubic spline interpolation, and a linear

interpolation, applied to data points of FT application. We observe *linear spline interpolation* and *cubic spline interpolation* fit our data better than the polynomial curve fitting techniques. Since the linear spline interpolation technique is simpler than cubic spline interpolation technique, we use linear interpolation for all our experiments. The fitted curve represents the CPI profile of that subroutine, we call this as a *subroutine signature*. For the subsequent iterations of the application, we use the subroutine signature of each subroutine to represent its execution. As a result, we are able to reduce the simulation time of the application by reducing the number of iterations to be simulated using Simics while accurately predicting the CPI behavior. Figure 6 illustrates a simple example of how the subroutine calls are represented in the original application and in our CFBS model.

5 Experimental Evaluation

In this section, we present our experimental results. Consider Figure 7, which shows the CPI behavior of FT benchmark on a four-core processor configuration. The left plot shows the CPI metric as reported by Simics, the middle plot shows the CPI metric as reported by MCCS, and the right plot shows the CPI metric as reported by CFBS. We observe that the CPI behavior of CFBS is much closer to Simics than MCCS. In Figure 13, we show the average relative error in CPI for MCCS and CFBS models compared to Simics. For FT benchmark, these errors are 90% and 2% respectively. In this case, the inaccuracy in MCCS can be attributed to lesser number of instructions simulated using Simics before switching to our Monte Carlo predictive model. Figure 8 shows results for Simics, MCCS and CFBS methods for BT benchmark. The average relative error in CPI for MCCS model is around 42%. The other observations and results for BT are similar to that of FT. Figures 9, 10, 11 and 12 show the CPI behavior reported by these models for CG, Barnes, Ocean and Raytrace benchmarks respectively. We can see from Figure 13 that apart from FT and BT, MCCS model's average relative error compared to Simics is around 20%. Overall, we see that the MCCS and CFBS schemes are 68% and 97.4% accurate on average. The higher accuracy of CFBS is due to the repetitive behavior of our target applications.

In Table 3, we list the simulation times, in minutes, for our benchmarks using Simics, MCCS, and CFBS. Note that under MCCS, values within the parenthesis are the simulation times of pure Simics and our Monte Carlo predictive model, respectively.

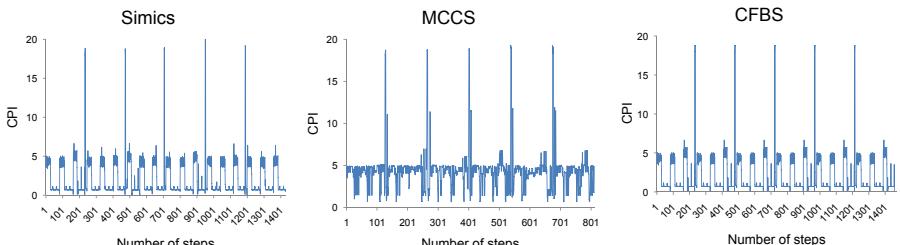


Fig. 7. CPI behavior of FT when run on a four-core processor

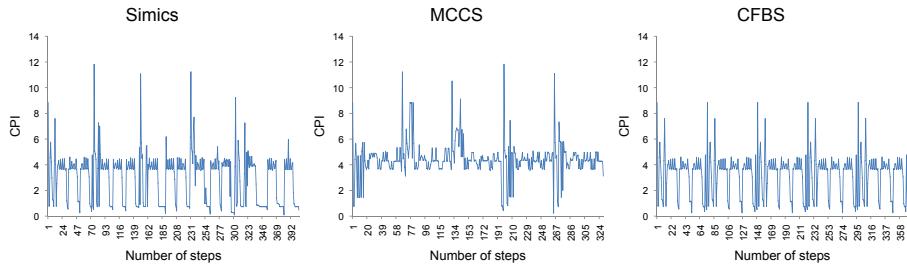


Fig. 8. CPI behavior of BT when run on a four-core processor

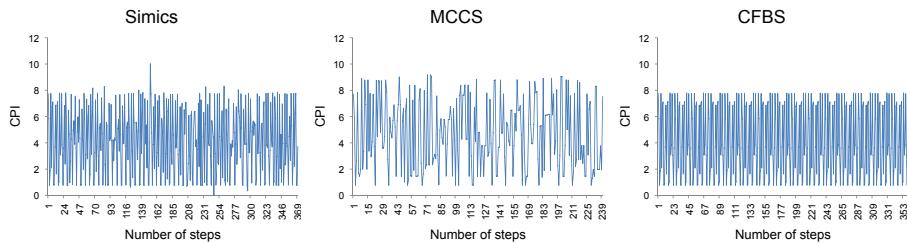


Fig. 9. CPI behavior of CG when run on a four-core processor

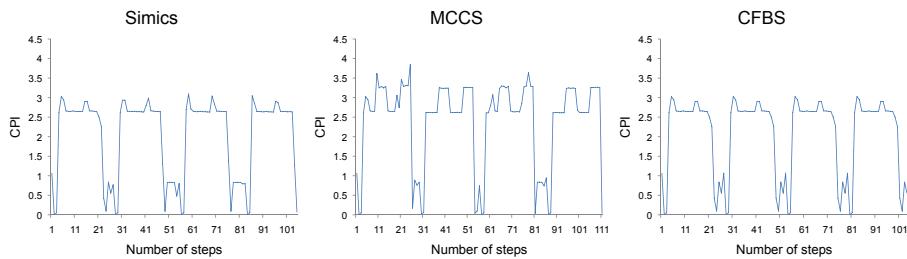


Fig. 10. CPI behavior of Barnes when run on a four-core processor

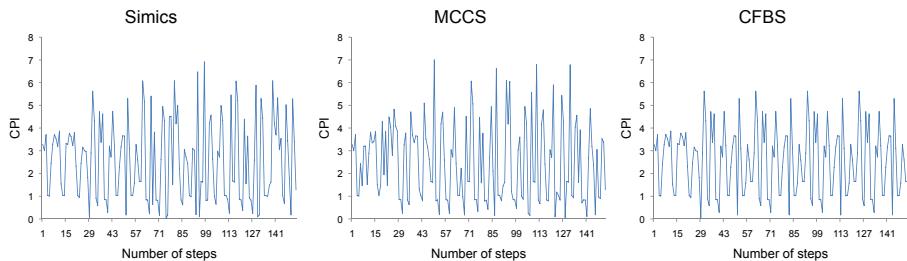


Fig. 11. CPI behavior of Ocean when run on a four-core processor

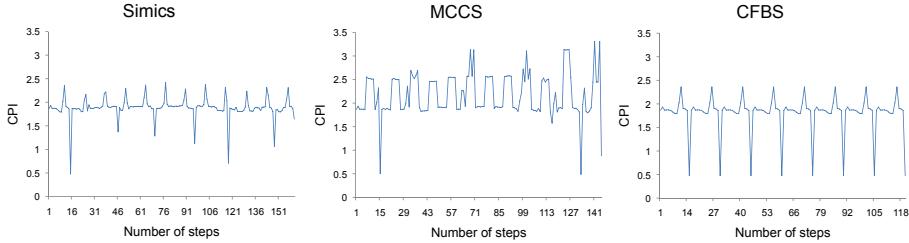


Fig. 12. CPI behavior of Raytrace when run on a four-core processor

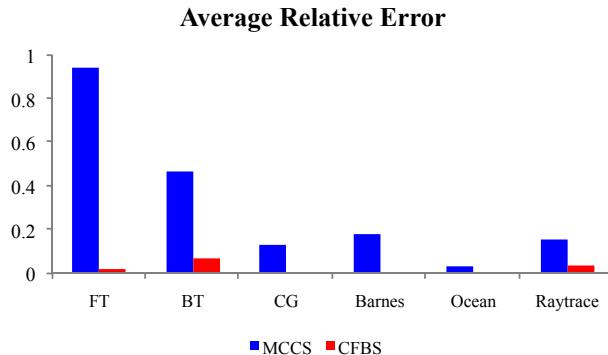


Fig. 13. Average relative error of benchmarks with respect to Simics

We see from Table 3 that CFBS takes the least time to simulate these benchmarks. We also observe that simulation time of MCCS is dominated by the execution within Simics. For CFBS, the simulation time is calculated as the time required to run one iteration of the application using Simics, as the rest of the simulation is done using curve fitting. In Table 4, we report the speedup of our techniques over Simics. We observe that CFBS model reduces the simulation time by as much as 13.6 times and 6.42 times on average while MCCS model reduces the simulation time by as much as 2.37 times and 1.77 times on average.

Table 3. Time taken by simulators in minutes

Benchmark	Simics	MCCS	CFBS
FT	1,245	524 (470 + 54)	190
BT	530	320 (280 + 40)	110
CG	410	176 (160 + 16)	30
Barnes	105	74 (70 + 4)	25
Ocean	113	100 (98 + 2)	55
Raytrace	170	104 (80 + 24)	25

Table 4. Speedup with respect to Simics

Benchmark	MCCS	CFBS
FT	2.37	6.55
BT	1.65	4.8
CG	2.32	13.6
Barnes	1.41	4.2
Ocean	1.13	2.05
Raytrace	1.63	6.8

6 Sensitivity Study

In this section, we conduct a sensitivity study, on the impact of the number of cores and the instruction window size, on the accuracy of the CPI prediction. In our first study, we consider the impact of changing a four-core base processor configuration to a one-core processor configuration and an eight-core processor configuration. In our second study, we increase the instruction window size to 20 million instructions and observe its impact on prediction accuracy.

For our first sensitivity study, we simulate FT and Raytrace benchmarks on one-core and eight-core processor configurations. Consider Figure 14, the plots on the left and the center show the average relative error in CPI prediction for these benchmarks for a one-core and eight-core processor configurations respectively. We observe that CFBS method has a good prediction accuracy for both the benchmarks on one-core and eight-core processor configurations. Notice that the prediction accuracy of MCCS for a one-core processor is much higher than that for a eight-core processor for FT benchmark. We attribute this higher prediction accuracy to a more consistent CPI behavior of FT within its phases for a one-core processor. For FT, the accuracy of MCCS model is much better for one and eight-core processors when compared to a four-core processor. We believe this difference in accuracy is due to the way in which FT behaves on one, four and eight-core processors. The behavior of FT on one and eight-core processors is favorable for MCCS as we are able to easily capture the sub-phase CPI behavior. We note that by having more sub-phases (interleavings), we would be able to get better accuracy for four-core processors as well. The CFBS method has higher accuracy for one and eight-core processors for both FT and Raytrace as these applications have repetitive codes and our CFBS method is able to capture this repetitive behavior.

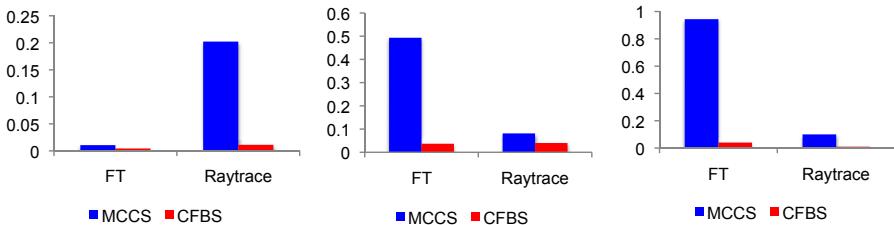


Fig. 14. Average relative error of benchmarks with respect to Simics for a **Left:** one-core processor. **Center:** eight-core processor. **Right:** four-core processor when instruction window size is set to 20 million instructions.

For our second sensitivity study, we change the size of instruction window from 10 million instructions to 20 million instructions. The rightmost bar-chart in Figure 14 shows the average relative error for FT and Raytrace benchmarks for a four-core processor configuration. We observe that there is not much impact on the CPI behavior prediction accuracy of our models by increasing the instruction window size. We can observe this by comparing the right plot in Figure 14 with Figure 13.

7 Related Work

For over a decade, computer architects have used software simulation techniques to evaluate the future computer hardware. SimpleScalar [3] was one of the earliest simulators used to simulate uniprocessor superscalar architectures. Some of the other flavors of uniprocessor simulators used are PTLsim [24], a cycle accurate x86 microprocessor simulator and SIMCA [16], a simulator for multithreaded computer architectures.

Recent advances in multicore architectures have led researchers to use simulators that can model multicore architecture. SESC [5], is a cycle accurate multicore simulator, while Simics [10], is a full system simulator that can be used to simulate multicores. GEMS [9] is a Simics based multiprocessor simulator with focus on accurate performance modeling. Due to the slow and complex nature of these cycle accurate and full system simulators, new statistical and analytical performance modeling techniques have emerged. Three such popular simulation techniques are SMARTS [11], Simflex [13], and Simpoint [4]. SMARTS framework applies statistical sampling to microarchitecture simulation. It samples and simulates a subset of benchmark's instructions to estimate the performance of the entire benchmark. Simflex use Simics to provide functional simulation and applies SMARTS methodology to do the sampling. It leverages Simics's checkpointing capability to store the state of simulation. A large checkpoint library is created by checkpointing at various points during the execution of the entire program in Simics. Such libraries are required for every *<simulated configuration, benchmark>* pair. These libraries are sampled to find the checkpoints that needs to be simulated. Although Simflex is statistically rigorous and accurate, it is rigid in terms of simplicity of use and requires a lot of memory to store the checkpoint libraries. We believe that for good quick estimate of performance, our techniques, CFBS and MCCS are more suited as they are easy to implement and have no extra memory requirements. SimPoint uses offline algorithms (clustering techniques) to detect phases in a program. This classification helps to choose simulation points that is representative of the phases, thereby, reducing the overall simulation time. This approach is independent of the architecture on which the program is run. We believe that it would be a challenge to use SimPoint for multicore simulation of applications (like FT) that have different runtime behavior based on the number of cores on which it is run.

Other simulation techniques to speedup the simulation include, HLS [8], a hybrid simulator that uses statistical profile of applications to model instruction and data streams, and MonteSim [23][22], a predictive Monte Carlo based performance model for in-order microarchitectures. However, these simulators were developed for uniprocessor architecture. Our methods differ from HLS in the way we profile applications, are generic and can be applied to multicores. Our predictive model is similar to [23] in some aspects like the use of Monte Carlo technique, however unlike MonteSim, we can use our methods to model multicore processors.

8 Conclusions and Future Work

We have presented two hybrid models, MCCS and CFBS, to address the challenge of fast evaluation of design-tradeoffs for multicore architectures. Our experimental analysis indicates that MCCS can reduce simulation time by as much as a factor of 2.37, with

a speedup of 1.77 on average compared to Simics. However, its average relative error is rather large at 32%. The results also reveal that CFBS can reduce simulation time by as much as a factor of 13.6, with a speedup of 6.24 on average. Additionally, the observed average relative error in CPI compared to Simics is significantly less at 2%.

Our results show that CFBS performs consistently better than MCCS in terms of both accuracy and speedup. One reason for this is our target application domain, namely, scientific applications. Since most of these applications have repetitive (iterative) codes, CFBS performs better as it is able to capture the entire phase behavior while MCCS only predicts the entire phase behavior based on partial phase results. Although MCCS performs worse in both speedup and accuracy, it has the potential of being more generic than CFBS model. For example, if the CPI behavior of a subroutine varies across different iterations, we believe that MCCS method would perform better than CFBS method. As part of future work, our initial plan is to investigate non-scientific applications to test the applicability of these methods.

References

1. From a few cores to many: A tera-scale computing research overview. Technical report, Intel Teraflops research chip, <http://techresearch.intel.com/articles/Tera-Scale/1449.htm>
2. Austin, T., Larson, E., Ernst, D.: SimpleScalar: An infrastructure for computer system modeling. *Computer* 35(2), 59–67 (2002)
3. Perelman, E., et al.: Using simpoint for accurate and efficient simulation. *SIGMETRICS Perform. Eval. Rev.* 31(1), 318–319 (2003)
4. Renau, J., et al.: SESC simulator (January 2005), <http://sesc.sourceforge.net>
5. Lauterbach, et al.: Ultrasparc-iii: a 3rd generation 64 b sparc microprocessor. In: ISSCC 2000. IEEE International on Solid-State Circuits Conference, 2000. Digest of Technical Papers., pp. 410–411 (2000)
6. Rosenblum, M., et al.: Complete computer system simulation: the simos approach. *IEEE Parallel and Distributed Technology: Systems and Applications* 3(4), 34–43 (Winter 1995)
7. Oskin, M., et al.: Hls: combining statistical and symbolic simulation to guide microprocessor designs. *SIGARCH Comput. Archit. News* 28(2), 71–82 (2000)
8. Martin, M.M.K., et al.: Multifacet’s general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News* 33(4), 92–99 (2005)
9. Magnusson, P.S., et al.: Simics: A full system simulation platform. *Computer* 35(2), 50–58 (2002)
10. Wunderlich, R.E., et al.: Smarts: accelerating microarchitecture simulation via rigorous statistical sampling. *SIGARCH Comput. Archit. News* 31(2), 84–97 (2003)
11. Woo, S.C., et al.: The splash-2 programs: characterization and methodological considerations. In: ISCA 1995: Proceedings of the 22nd annual international symposium on Computer architecture, pp. 24–36 (1995)
12. Wenisch, T.F., et al.: Simflex: Statistical sampling of computer system simulation. *IEEE Micro.* 26(4), 18–31 (2006)
13. Sherwood, T., et al.: Automatically characterizing large scale program behavior. In: ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems, pp. 45–57. ACM, New York (2002)
14. Heath, M.T.: Scientific computing: An introductory survey (2002)

16. Huang, J., Lilja, D.: An efficient strategy for developing a simulator for a novel concurrent multithreaded processor architecture. In: MASCOTS 1998: Proceedings of the 6th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, Washington, DC, USA, p. 185. IEEE Computer Society, Los Alamitos (1998)
17. Kahle, J.: The cell processor architecture. In: MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture, p. 3 (2005)
18. NASA. Nas benchmark suite, <http://www.nas.nasa.gov/Resources/Software/npb.html>
19. Pai, V.S., Ranganathan, P., Adve, S.V.: Rsim: Rice simulator for ilp multiprocessors. SIGARCH Comput. Archit. News 25(5), 1 (1997)
20. Intel Core Duo processor Frequently Asked Questions, <http://www.intel.com/support/processors/mobile/coreduo/sb/CS-022131.htm>
21. UltraSPARC T1 Niagara Specifications, <http://www.sun.com/processors/UltraSPARC-T1/specs.xml>
22. Srinivasan, R., Cook, J., Lubeck, O.: Ultra-fast cpu performance prediction: Extending the monte carlo approach. In: SBAC-PAD 2006: Proceedings of the 18th International Symposium on Computer Architecture and High Performance Computing, Washington, DC, USA, pp. 107–116. IEEE Computer Society, Los Alamitos (2006)
23. Srinivasan, R., Lubeck, O.: Montesim: a monte carlo performance model for in-order microarchitectures. SIGARCH Comput. Archit. News 33(5), 75–80 (2005)
24. Yourst, M.T.: Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In: IEEE International Symposium on Performance Analysis of Systems Software, 2007. ISPASS 2007, April 2007, pp. 23–34 (2007)

PSINS: An Open Source Event Tracer and Execution Simulator for MPI Applications

Mustafa M. Tikir, Michael A. Laurenzano, Laura Carrington, and Allan Snavely

Performance Modeling and Characterization Lab

San Diego Supercomputer Center

9500 Gilman Drive, La Jolla, CA

{mtikir,michaell,lcarrington,allans}@sdsc.edu

Abstract. The size of supercomputers in numbers of processors is growing exponentially. Today's largest supercomputers have upwards of a hundred thousand processors and tomorrow's may have on the order one million. The applications that run on these systems commonly coordinate their parallel activities via MPI; a trace of these MPI communication events is an important input for tools that visualize, simulate, or enable tuning of parallel applications. We introduce an efficient, accurate and flexible trace-driven performance modeling and prediction tool, PMaC's Open Source Interconnect and Network Simulator (PSINS), for MPI applications. A principal feature of PSINS is its usability for applications that scale up to large processor counts. PSINS generates compact and tractable event traces for fast and efficient simulations while producing accurate performance predictions. It also allows researchers to easily plug in different event trace formats and communication models, allowing it to interface gracefully with other tools. This provides a flexible framework for collaboratively exploring the implications of constantly growing supercomputers on application scaling, in the context of network architectures and topologies of state-of-the-art and future planned large-scale systems.

Keywords: High Performance Computing, Message Passing Applications, Performance Prediction, Trace-Driven Simulation, and Supercomputers.

1 Introduction

Performance models are calculable expressions that describe the interaction of an application with the computer hardware providing valuable information for tuning of both applications and systems [1]. An ongoing trend in High Performance Computing (HPC) is the increase in the total system core count; this in turn has permitted application scaling to tens and even hundreds of thousands of cores in recent years enabled by performance models that are used to guide application tuning [2-4]. Application performance is a complex function of many factors such as algorithms, implementation, compilers, underlying CPU architecture and communication (interconnect) technology. However as applications scale to larger CPU counts, the interconnect becomes a more prevalent factor in their performance requiring improved tools to measure and model them.

We present an efficient, accurate and flexible trace-driven performance modeling tool, PMaC's Open Source Interconnect and Network Simulator (PSINS), for MPI applications. PSINS includes two major components, one for collecting event traces during an application's run (*PSINS Tracer*), and the other for the replay and simulation of these event traces (*PSINS Simulator*) for the modeling of current and future HPC systems. The key design goals for PSINS are 1) scalability 2) speed 3) extensibility. To meet the first goal, PSINS Tracer runs with very low overhead to generate compact traces that do not use more bits than needed for a complete record of events. To meet the second goal, PSINS Simulator enables replay of events faster than real-time (a replay does not normally take as long as the original application run) while still producing accurate performance predictions. To meet the third goal, both PSINS Tracer and Simulator are provided freely as open-source, and have, in addition to its built-in trace formats, format conversion modules, and communication models, a graceful API designed such that anyone can easily extend these tools via plug-in virtual functions. PSINS interacts gracefully with other popular tracers and modeling and visualization tools such as that presented by Ratn et al. [5], MPIDtrace [6], Di-memas [7], TAU [8] and VAMPIR [9]. Figure 1 shows the high-level design of PSINS as well as the flow of information that occurs for performance prediction.

1.1 Tracer for Collecting Event Traces

PSINS provides a tracer library based on MPI's profiling interface (PMPI) [10]. PMPI provides the means to replace MPI routines at link time allowing tool developers to include additional instrumentation code around the actual MPI calls. The PSINS tracer library provides wrappers that serve as replacements for the MPI routines in the code (i.e. communication or synchronization events). For each MPI routine replacement, it

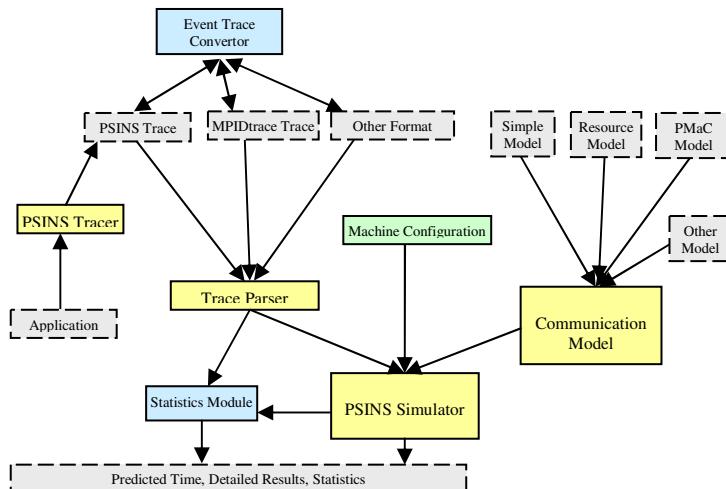


Fig. 1. The high-level design of PSINS and the flow of information within

uses additional code to gather detailed information about the called MPI function and its arguments. The tracer also gathers the time in between individual communication events or the computation time, labeled as *CPUBurst*. To gather CPUBurst events, the library uses timers at the end and the beginning of each MPI routine replacement so that when an MPI function is called, the time spent since the end of the last MPI call to the current call is recorded in the trace.

Since HPC applications typically run for long duration and tend to execute millions of MPI function calls, recording each event to a trace file as it occurs is not practical since this would introduce many small, latency-bound file I/O accesses. Like other efficient tracing tools [11, 28-29], PSINS Tracer uses per-task local memory buffers to temporarily store event information and only dumps the events in a task's buffer when that buffer is full. Moreover, to eliminate the need for any synchronization or any additional communication among tasks during tracing, initially PSINS generates a separate event trace file for each MPI task.

In a post-trace phase, to combine these separate trace files in to a single compact trace file, PSINS includes a trace consolidation utility, *mpi2psins*. The combining and compacting step is done serially after the execution of the traced application. This *mpi2psins* utility uses an encoding mechanism similar to general UTF encodings [12] in order to reduce the size of the final trace. It uses the most significant bit in each byte to determine the number of bytes that will be used to represent a number and the other seven bits to store the actual value. Using this technique it is possible to represent 2^n possible values with n bytes. An event trace is made up mostly of small integers that represent processor IDs, larger integers that represent message sizes, and real numbers that represent times. On average our encoding saves 60% of the size that would be required if these values were kept as normal 4 byte or 8 byte values. The trace thus serves as a minimal complete representation of events to which further compression techniques such as those that detect and encode regular expressions can be applied [26]. More importantly, when carrying out strong scaling studies, the size of communication traces encoded by this method grows linearly as function of processor count *even though the global communications may grow exponentially* [27]. This is because the time becomes shorter (at least for scalable codes) and the message sizes tend to decrease, and thus the UTF encodings become smaller with increasing processor count even though the total number of communications may go up.

Besides tracing functionality, PSINS tracer provides two additional libraries for performance measurement and analysis that can be included in the event trace run or collected independent from the trace. The first, called *PSINS Light*, is a library to measure overall execution time of the application and gather some event counts from the performance monitoring hardware (using PAPI [15]) in the underlying processors such as FLOP rate and overall cache miss counts. The second, called *PSINS Count*, is a library to measure the execution times and frequencies of each MPI function in the application in addition to those values collected by PSINS Light. PSINS Count is similar to IPM [14] and provides only a subset of information IPM provides.

1.2 Adding a New Input Trace Parser

In PSINS, the trace parser module is included as a separate module to allow the simulator to use different input trace formats easily. This allows users to easily add another

trace format such as TAU in addition to the already included parsers for PSINS and the MPIDtrace trace formats. A trace consists of a sequence of events that occur for each task and to use another trace format, the new parser needs only to convert events in the trace file to the PSINS internal representation of trace events.

In PSINS a new trace parser is added via use of virtual C++ functions. PSINS provides a base class, *Parser*, with a few virtual methods (see technical report [13] for more detail), which provide minimal functionality to access and consume the trace. Even though adding new parsers to PSINS requires some coding knowledge, PSINS hides most of the complexity of this process by providing most of the common infrastructure that is used by all parsers, requiring only the implementation of a few virtual methods. For example the parser for PSINS built-in trace format requires only 384 lines and the parser for MPIDtrace format requires 647 lines of C++ code.

1.3 Simulator for Performance Prediction

PSINS Simulator takes the communication event trace for an application and a set of modeling parameters for the target system and then replays the event trace for the target system, essentially simulating the execution of the parallel application on the target system. To simulate an MPI application on a target system, PSINS models both computation and communication times for each task in the application. To simulate an execution on a target system, the simulator needs details about the configuration and construction of the system. These modeling parameters consist of configurable components of a parallel HPC system.

PSINS assumes that the target architecture is a parallel computer composed of multiple computation nodes connected via configurable number of global busses (as shown in Figure 2). Each computation node contains a configurable number of processing units (processors or cores) and incoming and outgoing links to the global busses. It provides the flexibility for each compute node to have different numbers of incoming and outgoing links to the global busses and different number of processing units in the node. In addition, the processing units within a compute node can be specified to have different speeds. By using a flexible description of the target system architecture, PSINS provides the capability to simulate varying types of systems ranging from computational grids to shared memory multiprocessor systems.

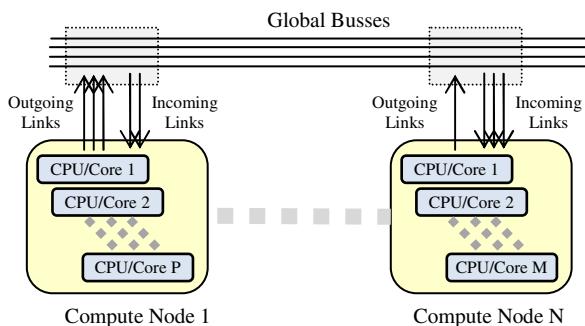


Fig. 2. Target architecture for simulation

All of these configurable modeling parameters are given to simulator in a small ASCII configuration file. The configuration file contains parameters for the system as a whole, for each compute node and for the MPI task-to-processor mapping. For the system, required parameters include the number of compute nodes, the best achievable bandwidth and latency for each bus for two nodes to communicate, and the number of busses. For each compute node, required parameters include the number of processing units, the number of incoming and outgoing links from/to the busses, the best achievable local bandwidth and latency within the node, a mapping of MPI tasks to processors, and CPU ratios which describe relative speeds (ratios) for the computational work of the target with respect to the base system. This CPU ratio is used by PSINS to model the computation time. Modeling done by simply projecting the time spent for each CPU Burst event to the target system using this ratio of how much faster or slower the processing unit in the target system is relative to the base system. This approach has been shown to be effective in previous research [1,6].

PSINS Simulator consumes events from the input trace in the order of their occurrence. The simulator uses an event queue based on priority queues to replay the input trace. When an event is read, it is tagged with the earliest time it will be ready for execution as its priority. This time is the value of the per-task timer at the time of insertion for the task that event belongs to. If an event is not ready for execution such as a blocking receive, or global communication, it is re-inserted into the event queue for later processing with its priority reduced. An event is deleted from queue when its execution is over. When an event is executed, it is marked with its execution time as well as its wait time. The wait time is a record of time the event had to wait for its execution as in imbalanced parallel applications with blocking communications or barriers. After its execution, the execution timer for its task is incremented accordingly and global timer is updated for synchronous simulation.

The execution of an event during simulation depends on the type of the event and the state of the system at each event execution. The state of a system at any given time is a combination of the best achievable bandwidths and latencies, the bus load, contention, traffic in the network and the underlying network topology. If it is a CPU burst event, it is completed by calculation of its time on the target system using the CPU ratio described above. For blocking communication events, it is kept in the queue until its mate is posted. If the event is a global communication, it is kept in the queue until all participating tasks post the same event. When all participating tasks post the event for the communication, communication model is asked to calculate the bandwidth and latency at the time of its execution and the event is executed.

PSINS Simulator includes a statistics module to collect detailed information about the simulation of an event trace on the target system, similar to IPM. The statistics module collects information about the event execution frequencies, computation and communication times for each task as well as the execution time for each event type. It also collects the waiting time for each event type to provide information on load balancing. Moreover, it generates histograms on message sizes and on the ranges of bandwidths calculated by the communication model for the communication events.

Such information provides valuable feedback to users and developers to help them understand the interaction of applications with the target system, and can be valuable to guiding optimization efforts for the application. More importantly, this information is useful for verifying simulation accuracy by comparing it to the same information measured during an actual run on the target system.

2 Communication Models

PSINS isolates the modeling parameters and communication models from the simulator (as shown in Figure 1) to enable users to easily investigate new communication models. From the perspective of the PSINS Simulator, the communication model is a black box. By separating the parameters for the target system from the communication model, PSINS allows even more flexibility toward investigating the impact of different communication models.

The communication model takes an event, the parameters from the configuration file, and the current state of the simulated system to calculate the sustained latency and bandwidth for the messages that are associated with that event. The model is responsible for determining when an event will be executed, which might be at some point in the future due to the unavailability of resources or some other measure of contention. The model also determines which resources it will require and for how long the resources are required, which in turn can change the state of the simulated system based on the needs of the event. The communication model then calculates the time to complete the event including the time to transmit a message as well as the time that the message must wait for resources (wait time). Moreover, each event can have its own model. These models can be simple (i.e. based on bandwidth and latency) or more complex functions of the system's state, the number of processors involved in the event, and the scalability of the event on the network.

2.1 Built-In Models

PSINS includes several built-in communication models that can be used to investigate a target system. These models are the *simple* model, the *resource contention* models, and the *PMaC* model. Our experience [16] indicates that these models can accurately be used to model application performance for a majority of today's HPC systems.

The simple model uses the best sustainable bandwidth and latency from the configuration file and assumes the resources available to the system are infinite. That is, when a message is ready to be sent, it assumes that resources along the path of the message are available and calculates the time to send the message as a simple addition of latency to the time spent to transfer the message body. For collective communications, this model uses a simple description for each communication event that indicates whether that event scales in *linear*, *logarithmic* or *constant* time with respect to the number of participating tasks. The simple model is designed to model the lower bound for the communication time for an application.

PSINS provides three resource contention models based on the number of global busses, incoming, and outgoing links from compute nodes, called *bus-only*, *incoming-link-only*, and *outgoing-link-only* models. These models assume that the number of a certain type of resource that is available for communication is limited and use a scheduling algorithm to schedule each message based on resource availability. These models are designed to investigate the impact of resource contention on the performance of an application. For instance, by predicting the performance of an application for an increasing number of busses, users can get a feel for how sensitive the application's performance is to number of busses available, which in turn can identify whether the application posts multiple messages at around the same time.

In addition to simplistic models, PSINS also includes a more complex communication model, called the *PMaC* model. This model is more complex than the previous models in order to increase the accuracy of the simulations. For point-to-point communications, this model takes the number of outstanding messages at the time of a message delivery and, based on the current load on the busses and input and output links, scales the maximum bandwidth accordingly.

For collective communications, alternative to using simple description of each MPI collective communication routine, the *PMaC* model also provides the means to use a more complex and realistic bandwidth calculations based on message sizes. This is done by measuring the bandwidth for each collective communication routine for an increasing size of messages using the synthetic benchmark, *PSINSBench*, that is included in PSINS package (see technical report [13] for details). Then using a curve-fitting algorithm the measured bandwidths are fit to a continuous function, which is later used by the model to calculate the bandwidth for a given message size.

2.2 Adding a New Model

In addition to the built-in models, PSINS allows users to easily plug-in new communication models. Like trace parsers, new communication models are added with virtual C++ functions. PSINS provides a base class, *Model*, with some virtual methods (see [13] for the list of virtual functions). These virtual methods provide the functionality to schedule events on resources as well as to calculate the time it takes to execute an event. Then, to create a new communication model, the developer needs only to define a class that extends the *Model* class and implement its virtual functions.

Much of the burden of the model developer then resides in the areas that are almost completely model-specific, which leaves only a few virtual functions for the developer to implement. Among the built-in models in PSINS, the simplest model requires 228 lines of C++ code. A collection of resource contention models requires 158 lines of C++ code and the most complex model requires 433 lines of C++ code.

3 Experimental Results

To demonstrate the usability, efficiency and accuracy of PSINS Tracer and Simulator, we have conducted several experiments where we used PSINS Tracer to collect MPI event traces for three scientific applications: AVUS [17], HYCOM [18] and ICEPIC [19] from the TI-09 Benchmark Suite [20].

All of the PSINS traces were collected on a base system, NAVO's IBM Cluster 1600 (3072 cores connected with IBM's High Performance Switch), called *Babbage*. We ran each scientific application with two of their input data sets, namely *standard* and *large*, with processor counts ranging from 59 to 1280. The actual runtimes for the applications range from 0.5 to 2.5 hours where each application runs for around half an hour at the highest processor count and was scaled to that count using the same input data set (i.e. strong scaling). For simulation of the collected traces, we ran the simulator on a Linux box with two dual-core processors. In addition to simulating the base system *Babbage*, we also simulated the MHPCC's Dell Cluster, called *Jaws* (5120 cores connected with Infiniband) and ERDC's Cray XT3 system, called *Sapphire* (8320 cores connected with Cray SeaStar engine). To compare PSINS to a

state-of-art simulation tool, we also collected MPI event traces using MPIDtrace[6] and simulated them using Dimemas[6] for each application and processor count. We present the results as event trace sizes, simulation times, and prediction accuracy.

3.1 PSINS Trace Sizes and Simulation Times

The sizes of PSINS traces collected for each application and processor count is given in Figure 3. The figure illustrates that the size of PSINS event traces grows linearly as the processor count grows. The sizes range from 4GB to 32GB and are at least 4 times smaller than the event trace sizes generated by a similar state-of-the-art MPI event tracer, MPIDtrace [6] (the sizes of traces from MPIDtrace are presented in [13]).

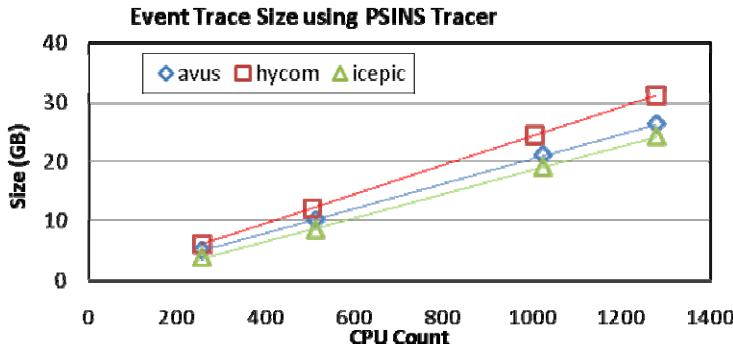


Fig. 3. PSINS event trace size vs. CPU count for 3 applications

These results suggest that one could practically store uncompressed traces for 10 thousand processors in about 300GB and for 100 thousand processor jobs in about 3TB using PSINS Tracer. Some compression techniques such as those used in [26] would be useful at large scale, though we note that some research groups already devote terabytes to storing the memory traces of strategic applications [25], so this same amount of storage devoted to communications traces is not out of the question.

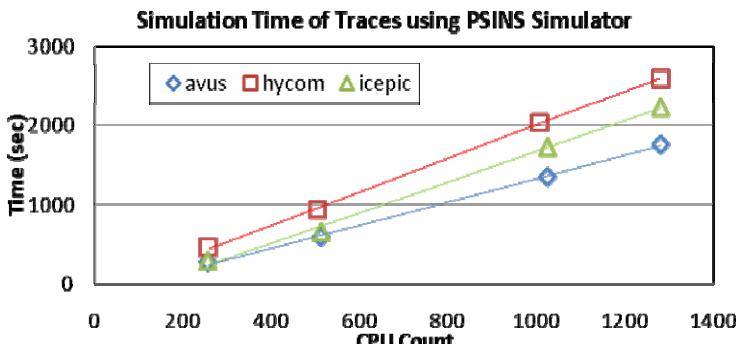


Fig. 4. PSINS Simulator simulation time vs. CPU count for 3 applications

These collected event traces were then fed through the PSINS Simulator, the simulation times are presented in Figure 4. The figure shows that PSINS Simulator is able to replay these collected traces for a target system in under 1 hour for all applications. On average the replay takes 7 times less time than running the program, however the replay time also grows linearly with processor count suggesting that in the future the replay procedure should itself be parallelized using the natural synchronization points that occur at global communications. However, these simulation times are already an order of magnitude faster than a similar network simulator Dimemas [6] (for a detailed comparison to Dimemas see [13]).

Combined, Figure 3 and Figure 4 show that for each application there is a linear correlation between the input trace size and the time it takes to replay the trace for a target system in PSINS. They also demonstrate that PSINS Tracer collects MPI event traces of manageable and tractable sizes and PSINS Simulator replays these traces in a tractable time for a target system. This indicates that as applications scale to even larger processors counts, PSINS is likely to continue to be usable and effective.

In addition to trace size and simulation time, it is also important to quantify the overhead introduced by the PSINS Tracer itself during trace collection even though the cost is only born once. During our experiments, we observed that the overhead of PSINS Tracer ranges from 0.2% to 14.8% compared to the original execution times of the applications, which is very similar to the overhead of the state-of-art tool MPIDtrace. The average overhead for all applications and processor counts is 5.9% meaning it can be efficiently used for large processor counts, even in production runs.

3.2 Simulation Accuracy

Even though the usability of PSINS in terms of event trace sizes and simulation efficiency and tracing overhead is important, what matters most is the accuracy of the predictions produced by the models. To investigate accuracy at a finer granularity, we simulated an event trace collected using PSINS Tracer for HYCOM with 124 processors for the base system and compared the communication times simulated to the measured times for each task. For this experiment we used the built-in simple communication model.

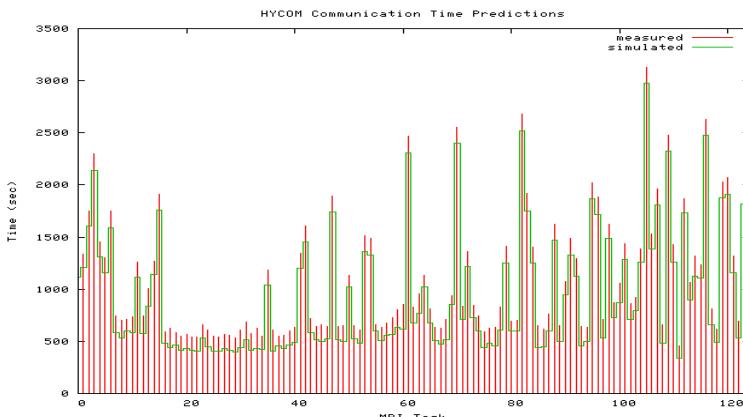


Fig. 5. Measured and simulated communication times for all tasks

Figure 5 presents the communication times measured and predicted for each task. The red vertical bars are used to represent the measured times whereas the green horizontal line is used to represent the simulated times. Figure 5 shows that PSINS Simulator is quite accurate in predicting the communication time for each task. The average absolute error in predicting the communication times for all tasks is 17% whereas the error in predicting the total communication time is 14%. More importantly, Figure 5 shows that despite the imbalance in communication times among tasks, the results of PSINS simulation closely match the observed behavior. Note again that the results in Figure 5 show simulation results using the built-in simple model, which tends to under predict the communication times.

In addition to comparing communication times for each task, we further broke down the communication time into the time spent in each MPI routine. Figure 6 (a) presents the measured values for the percentages of time spent in each MPI routine and Figure 6 (b) presents the percentages for the same MPI routines from the PSINS simulation. Figure 6 shows that the percentage of time spent in MPI routines from the simulation closely matches the percentages from the actual run, indicating that the simulation results match the measured results at an even finer granularity.

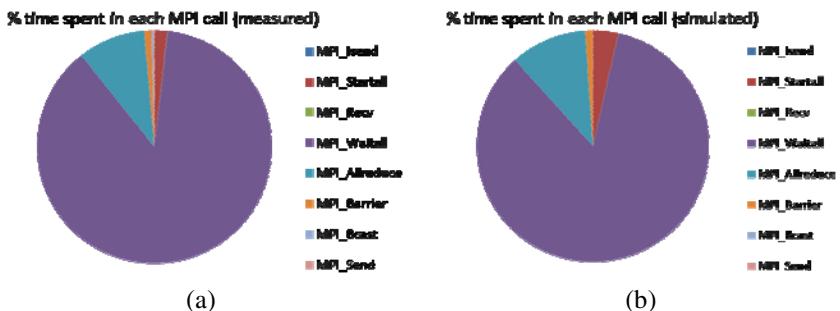


Fig. 6. Communication time spent in MPI calls for HYCOM

Table 1. Total time (sec) spent in communication events (using PMaC model)

	CPU Count	Jaws			Sapphire		
		Simulated	Measured	% Error	Simulated	Measured	% Error
HYCOM	124	121,476	128,285	-5%	161,055	167,620	-4%
HYCOM	504	449,646	519,335	-13%	573,365	621,793	-8%
AVUS	64	27,764	26,194	6%	30,680	22,561	36%
AVUS	1280	1,333,414	1,193,967	12%			
ICEPIC	64	72,144	71,073	2%	89,950	88,708	1%
ICEPIC	1280	1,178,914	1,142,970	3%			

Table 1 presents a comparison between the total communication times measured during an actual run and times simulated by PSINS for two HPC systems. We used the PMaC model for the simulations listed in this table; it shows the ability to predict the communication times of applications within 15% error for all cases except AVUS running with 64 processors on Sapphire. The absolute average error among all cases

is only 9.0%. Overall, Table 1 demonstrates that PSINS is effective in modeling and predicting the performance of applications for HPC systems. The largest error in communication time occurs in AVUS with 64 processors on Sapphire. However despite a large relative error, the communication time accounts for only 7% of the overall execution time and the runtime prediction error from Table 2 is only 2%.

Similarly, Table 2 presents the comparison between the execution times measured during actual runs and runtime predictions from PSINS (recall that event traces are collected on a different system than these HPC systems). In PSINS simulations, the relative speed of compute units in each target system to the base system is calculated using the PMaC Prediction Framework [25].

Table 2 shows that the absolute prediction error using PSINS is under 10% for majority of the cases and except 4 cases for Sapphire, it is under 15%. The prediction error ranges from -9.9% to 6.8% for Jaws (average absolute error is 7.4%) and it ranges from -26.3% to 18.1% for Sapphire (average absolute error is 11.6%). Further investigation has shown that error in the relative speed calculation of its compute units is largely responsible for the higher error for the Sapphire predictions. Table 2 demonstrates that PSINS is effective in modeling and predicting the overall execution times of applications on HPC systems as well as the total communication times.

Table 2. Simulated (using PMaC model) and measured runtimes (sec)

	Input Deck	CPU Count	Jaws			Sapphire		
			Runtime (sec)		% Error	Runtime (sec)		% Error
			Simu.	Meas.		Simu.	Meas.	
HYCOM	STD	124	3,336	3,243	2.9	3,439	4,282	-19.7
		501	1,113	1,042	6.8	1,309	1,128	16.0
	LRG	256	5,973	5,800	3.0	5,756	5,956	-3.4
		504	2,816	3,002	-6.2	2,764	3,752	-26.3
AVUS	STD	64	7,062	7,835	-9.9	7,934	7,835	1.3
		384	1,366	1,293	5.6	1,619	1,371	18.1
	LRG	512	3,394	3,768	-9.9	3,721	4,018	-7.4
		1280	1,850	1,769	4.6			
ICEPIC	STD	64	4,284	4,185	2.4	5,419	5,082	6.6
		384	2,237	2,600	-13.9	2,212	2,086	6.0
	LRG	512	2,251	2,563	-12.2	2,929	2,623	11.7
		1280	2,158	2,420	-10.8			

4 Related Work

Early work on performance prediction of HPC applications was done in the Proteus simulator [21], an execution-driven simulator which met many of the design goals that have been laid out for PSINS at the time. Proteus was designed modularly so that it could be customized for the target system and tradeoffs could be made between accuracy and efficiency by using a different implementation of a certain simulation component. Unfortunately Proteus introduces a slowdown of 2-35x for each process in the target application, which renders it cumbersome for the purpose of simulating long-running large-scale applications at thousand of processors.

Later work, such as Parallel Proteus [21], LAPSE [22], MPI-SIM [23] and the Wisconsin Wind Tunnel [24] improved the efficiency of the simulation required to make predictions by executing simulations in parallel. Typically these tools are execution-driven and perform parallel discrete event simulation and tend to be full machine simulators that address many aspects of a target architecture other than the network. This causes them to be slower and more complex and less modular than PSINS for the purpose of MPI scaling investigations.

The Dimemas project [7] uses the concept of largely divorcing network prediction from the prediction of serial computation portions of the code. Like PSINS, the user supplies Dimemas with a speedup ratio for a target system. Dimemas uses this speedup ratio along with the MPI event trace (in their case called an MPIDTrace) to perform a discrete event simulation of the application on a target system. Unlike PSINS, Dimemas is not open source, hence though useful it is not quite satisfactory as a medium for community development in this arena. Dimemas currently stores their MPI event traces as an ASCII text file resulting in large event traces files.

5 Conclusions

Performance models can provide valuable information in the tuning of both applications and systems, enable application-driven architecture design and extrapolate the performance of applications on future systems. In the constantly changing and growing field of HPC, it is important to have a modeling tool that is flexible enough to adapt to architectural changes and is scalable enough to grow with the constantly increasing system sizes. PSINS has this flexibility and scalability along with specific features that make it practical to use for model generation. PSINS tracer allows event traces to be captured with low overhead and recorded at manageable sizes even for large processor counts of MPI applications. PSINS simulator is capable of simulating different HPC networks with a high degree of accuracy in a reasonable amount of time. This makes PSINS a multifunctional tool of which flexibility, scalability, and accuracy allow its utilization in collaborative studies involving modeling large scale HPC applications.

PSINS Tracer and Simulator is already ported for several HPC systems and is available at <http://www.sdsc.edu/pmac/projects/psins.html>.

Acknowledgments. This work was supported in part by the DOD High Performance Computing Modernization Program and the DOE Office of Science through the SciDAC2 award entitled Performance Evaluation Research Center.

References

1. Bailey, D.H., Snavely, A.: Performance Modeling: Understanding the Present and Predicting the Future. In: EuroPar (2005)
2. Michalakes, J., Hacker, J., Loft, R., McCracken, M., Snavely, A., Wright, N., Spelce, T., Gorda, B., Walkup, R.: WRF Nature Run. In: Supercomputing (2007)
3. Alvarez, G., Summers, M., Maxwell, D., Eisenbach, M., Meredith, J., Larkin, J., Levesque, J., Maier, T., Kent, P., D'Azevedo, E., Schulthess, T.: New algorithm to Enable 400+ TFlop/s Sustained Performance in Simulations of Disorder Effects in High-Tc Superconductors, Gordon Bell Prize Winner. In: Supercomputing (2007)

4. Carrington, L., Komatitsch, D., Tikir, M.M., Laurenzano, M., Snavely, A., Michea, D., Tromp, J., Le Goff, N.: High-frequency Simulations of Global Seismic Wave Propogation Using SPECFEM3D_GLOBE on 62K Cores. In: Supercomputing (2008)
5. Ratn, P., Mueller, F., de Supinski, B., Schulz, M.: Preserving Time in Large-scale Communication Traces. In: Supercomputing (2008)
6. Badia, R., Labarta, J., Giménez, J., Escalé, F.: Dimemas: Predicting MPI Applications Behavior in Grid Environments. In: Work. on Grid Applications and Prog. Tools (2003)
7. Girona, S., Labarta, J., Badia, R.: Validation of Dimemas Communication Model for MPI Collective Operations. In: Proceedings of the European Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface (2000)
8. Shende, S., Maloney, A.: The TAU Parallel Performance System. International Journal of High Performance Computing Applications (2006)
9. Nagel, W., Arnold, A., Weber, M., Hoppe, H., Solchenbach, K.: VAMPIR: Visualization and Analysis of MPI Resources. Supercomputer 12(1), 69–80 (1996)
10. MPI Profiling Interface, <http://www mpi-forum.org/docs/mpi-11-html/node152.html>
11. Tikir, M.M., Laurenzano, M., Carrington, L., Snavely, A.: PMaC Binary Instrumentation Library for PowerPC/AIX. In: Workshop on Binary Instrumentation and Applications (2006)
12. Wikipedia contributors. UTF-8, <http://en.wikipedia.org/wiki/UTF-8> (accessed, 2009)
13. PSINS: An Open Source Event Tracer and Execution Simulator for MPI Applications, Extended Version, <http://www.sdsc.edu/pmac/projects/psins.html>
14. Integrated Performance Monitoring (2008), <http://ipm-hpc.sourceforge.net/>
15. Mucci, P., Browne, S., Deane, C., Ho, G.: PAPI: A Portable Interface to Hardware Performance Counters. In: Department of Defense HPCMP Users Group Conference (1999)
16. Tikir, M.M., Carrington, L., Strohmaier, E., Snavely, A.: A Genetic Algorithms Approach to Modeling the Performance of Memory-bound Computations. In: Lumpe, M., Vanderperren, W. (eds.) SC 2007. LNCS, vol. 4829. Springer, Heidelberg (2007)
17. Strang, W., Tomaro, R., Grismer, M.: The Defining Methods of Cobalt60: A Parallel Implicit, Unstructured Euler/Navier-Stokes Flow Solver, Institute of Aeronautics and Astronautics Paper 99-0786 (1999)
18. Bleck, R.: An Oceanic General Circulation Model Framed in Hybrid Isopycnic–Cartesian Coordinates. Ocean Modelling 4, 55–88 (2002)
19. Sasser, G., Blahovec, J., Bowers, L., Colella, S., Luginsland, J., Watrous, J.: Current Emission, Resistive Losses, and Other Challenging Problems in the Simulation of High Power Microwave Components, Inst. of Aeronautics and Astronautics Paper (1999)
20. Department of Defense, High Performance Computing Modernization Program. Technology Insertion (2009), <http://www.hpcmo.hpc.mil/Htdocs/TI/>
21. Brewer, E., Dellarocas, C., Colbrook, A., Weihl, W.: Proteus: A High-Performance Parallel Architecture Simulator, MIT Technical Report MIT/LCS/TR-516 (1991)
22. Dickens, P., Heidelberger, P., Nicol, D.: A Distributed Memory LAPSE: Parallel Simulation of Message-Passing Programs. In: Workshop on Parallel and Distributed Simulation (1994)
23. Prakash, S., Bagrodia, R.: MPI-SIM: Using Parallel Simulation to Evaluate MPI Programs. In: Proceedings of the Winter Simulation Conference (1998)
24. Reinhardt, S., Hill, M., Larus, J., Lebeck, A., Lewis, J., Wood, D.: The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In: Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (1993)

25. Snavely, A., Carrington, L., Wolter, N., Labarta, J., Badia, R., Purkayastha, A.: A Framework for Performance Modeling and Prediction. In: Supercomputing (2002)
26. Noetha, M., Ratna, P., Mueller, F., Schul, M., de Supinski, B.R.: ScalaTrace: Scalable compression and replay of communication traces for high-performance computing. *Journal of Parallel and Distributed Computing* (2008)
27. Kim, J.S., Lilja, D.J.: Characterization of communication patterns in message-passing parallel scientific application programs. In: Panda, D.K., Stunkel, C.B. (eds.) CANPC 1998. LNCS, vol. 1362, p. 202. Springer, Heidelberg (1998)
28. Gao, X., Simon, B., Snavely, A.: ALITER: An Asynchronous Lightweight Instrumentation Tool for Event Recording. In: Workshop on Binary Instrumentation and Applications
29. Gao, X., Laurenzano, M., Simon, B., Snavely, A.: Reducing Overheads for Acquiring Dynamic Traces. In: International Symposium on Workload Characterization, ISWC 2005 (2005)

A Methodology to Characterize Critical Section Bottlenecks in DSM Multiprocessors*

Benjamín Sahelices¹, Pablo Ibáñez², Víctor Viñals², and J.M. Llaberia³

¹ Depto. de Informática, Univ. de Valladolid
benja@infor.uva.es

² Depto. de Informática e Ing. de Sistemas, I3A and HiPEAC, Univ. de Zaragoza
{imarin,victor}@unizar.es

³ Depto. de Arquitectura de Computadores. Univ. Polit. de Cataluña
llaberia@ac.upc.edu

Abstract. Understanding and optimizing the synchronization operations of parallel programs in distributed shared memory multiprocessors (DSM), is one of the most important factors leading to significant reductions in execution time.

This paper introduces a new methodology for tuning performance of parallel programs. We focus on the critical sections used to assure exclusive access to critical resources and data structures, proposing a specific dynamic characterization of every critical section in order to a) measure the lock contention, b) measure the degree of data sharing in consecutive executions, and c) break down the execution time, reflecting the different overheads that can appear. All the required measurements are taken using a multiprocessor simulator with a detailed timing model of the processor and memory system.

We propose also a static classification of critical sections that takes into account how locks are associated with their protected data. The dynamic characterization and the static classification are correlated to identify key critical sections and infer code optimization opportunities (e.g. data layout), which when applied can lead to significant reductions in execution time (up to 33 % in the SPLASH-2 scientific benchmark suite). By using the simulator we can also evaluate whether the performance of the applied code optimizations is sensitive to common hardware optimizations or not.

1 Introduction

Shared memory multiprocessors use the memory as a means to synchronize and communicate processors. But, if several processors try to execute the same critical section concurrently by enforcing the required serialization, the scalability of parallel programs is often limited. When this happens, optimizing the lock ownership transfer among processors is essential for achieving high performance.

* This work was supported in part by Diputación General de Aragón grant “gaZ: Grupo Consolidado de Investigación”, Spanish Ministry of Education and Science grants TIN2007-66423, TIN2007-60625, Consolider CSD2007-00050, and the european HiPEAC-2 NoE.

To understand the opportunities for tuning critical sections, fine grain metrics such as lock time and time distribution inside the critical sections are required. Also, in order to identify and quantify bottlenecks in parallel applications, a precise control over the environment where the measures are taken is required. For these reasons, we use a multiprocessor simulator (RSIM) that models out-of-order processors, memory hierarchy and interconnection network in detail [12]. Within our simulation environment it is possible to take fine grain statistics, break down the execution time of critical sections accurately, or turn on/off architectural enhancements, which is difficult to accomplish in real systems. Unlike other simulation environments which generate a trace using a trace driven simulator and later need to define an interleaving of memory reference streams, our simulator already interleaves the memory reference streams [3].

In this work we suggest a two-step approach. In the first step we compute the execution time of all the critical sections, separating lock management and shared data access. We also characterize critical sections *dynamically* by quantifying two other important features: a) lock contention and, b) degree of data sharing. As a measure of lock contention we use the number of processors trying to access a lock variable when the lock is released, and to identify the degree of data sharing we measure the number of different shared data lines accessed in consecutive executions of the critical section. Finally, we further classify critical sections *statically* in two types, depending on how the data structure is protected by the lock variables. As we will see, we will combine the dynamic characterization and the static classification in order to identify the key critical sections and infer code optimization opportunities, such as data layout optimizations [456]. In this step the dynamic statistics are gathered in a Baseline system. The Baseline system has no architectural enhancements that might introduce noise in the measures, such as prefetching [7] or directory caching [89].

In the second step, we turn on the architectural enhancements and evaluate again the performance of the applied optimizations. From this second run we realize that the applied optimizations increase performance on the enhanced system as well.

The structure of the paper is as follows: Section 2 details the proposed characterization and classification procedure. Section 3 presents the simulation environment and the used benchmarks. In Section 4 we show the static classification of all SPLASH-2 critical sections. Then, in Section 5 we gather all the dynamic statistics, select and optimize the key critical sections, and compare the performance gains in both the Baseline and the enhanced systems. Lastly, the related work and conclusions Sections follow.

2 Static Classification and Dynamic Characterization of Critical Sections

This Section introduces first the static classification and then the execution time breakdown. Next we describe the twofold characterization of critical sections according to: a) lock contention and b) degree of data sharing. Finally, it is

explained how the methodology helps to detect and optimize code and data layout inefficiencies.

Static classification. By looking at the source code we classify critical sections into two types: in type A a single lock is used to protect access to a shared data structure, and in type B a lock array is used to protect fine-grained access to structured shared data. Each type is further split into two ad-hoc subtypes, 0 and 1. Inside the critical sections of type A.0 only a single shared variable is usually updated. Critical sections of type A.1 protect the access to several variables, for instance, when managing linked lists of shared-memory chunks. Type B.0 critical sections are used to protect access to task queues; here each processor has assigned a single task queue and only when this is empty the processor searches the task queues of other processors. Type B.1 identifies critical sections used to gain access to big data structures, where one entry of the lock array is used to protect the access to a data structure subset.

Time breakdown of critical section execution. Locks are used to assure exclusive access to shared data when executing a code section [10,11,12]. A processor requests the lock through a lock acquire operation (A) and frees it with a release operation (R), see Figure 1. In this paper we use a Test-and-Test&Set algorithm to acquire a lock. We call *lock time* the sum of acquire and release times, and critical section *Latency* the time between the end of the acquire operation and the beginning of the release operation. We can further split the acquire time into the waiting time (processor is spinning) and the hand-off time (it is being decided which processor enters next). Throughout the simulation we measure Lock time and Latency for each execution instance of every critical section.

When there is contention the Lock time is proportional to the Latency times the number of contending processors. So, small Latency reductions can have a large impact on the Lock time.

Lock contention. As a measure of lock contention we use the number of processors trying to access a lock variable when the critical section is released. The lock time of a critical section is directly related to its lock contention.

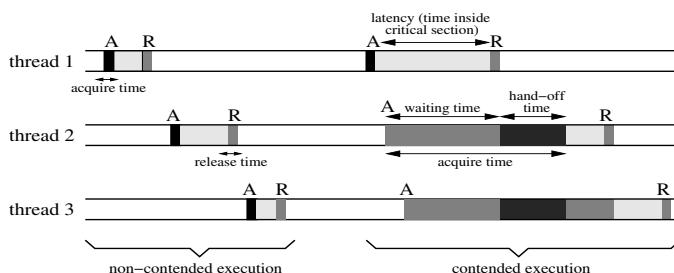


Fig. 1. Time breakdown of contended and non-contended execution of a critical section

Data sharing patterns. To measure the degree of data sharing we compare the data cache lines accessed in consecutive executions of a critical section. We distinguish three sharing patterns, namely *full-sharing*, *no-sharing*, and *some-sharing*. Full-sharing arises when a critical section protects the same line(s) in two consecutive executions. No-sharing arises when none of the lines accessed in a critical section has been used in the two previous executions. Finally, some-sharing collects the remaining execution instances that show an intermediate sharing pattern.

2.1 Detecting Inefficiencies and Inferring Optimizations

In order to choose the critical sections key to the overall performance we add, for all instances of each critical section, the Lock time and the Latency, and select those that represent a relative high fraction of the parallel execution time.

We use the Latency in each execution of a key critical section in order to infer the possibility of a data layout problem and then inspect the code. For instance, if the Latency is not biased towards a few values there could be false sharing. This is particularly true in the type A.0. Also, by analyzing the critical section Latency, bad programming habits can be detected. We will see some examples later.

When a contended lock shows low degrees of data sharing, the contention is not caused by the simultaneous access to the shared data, but by the simultaneous access to the lock variable. In other words, the same contended lock is used to protect the access to different shared data in consecutive executions. For instance, a programming technique to protect the access to complex data structures is using a lock array and a hash function to distribute the concurrently accessed elements of the data structure uniformly among the elements of the lock array. If some elements of the lock array have high contention and low data sharing, performance may be improved by using a better hash function, by increasing the lock array size or both.

Section 5 details all the detected inefficiencies and the feasible solutions. Anyway, the most often employed optimizations to tackle the Latency problem consist in reorganizing the data layout to either remove false sharing [4] or/and enforce collocation of the lock and the shared variables [5]. For type B.1 the action taken has been to increase the number of entries in the lock array (finer granularity protection).

3 Baseline System and Benchmarks

The runtime statistics are obtained with RSIM [12]. RSIM is an execution-driven simulator performing a detailed cycle-by-cycle simulation of an out-of-order processor, a memory hierarchy and an interconnection network. The simulator models all data movements including the cache port contention at all levels. The processor implements a sequential consistency model using speculative load execution [7]. The Baseline system is a distributed shared-memory multiprocessor with a MESI cache coherence protocol [13]. The interconnection is a

wormhole-routed, two-dimensional mesh network. The contention of ports, switches and links is also modeled accurately. Table II lists the processor, cache and memory system parameters as well as the tested SPLASH-2 benchmarks [14].

Table 1. Baseline system parameters and benchmarks

Processor	32 processors, 1 Ghz	L2/Memory Bus	Split.32 B,3 cycle+1 arbit.
ROB	64 entry, 32 entry LS queue	Memory	4 way interl., 100 cycle DRAM
Issue	out-of-order issue/commit 4 ops/cyc.	Directory	SGI Origin-2000 based MESI
Branch	512 entry branch predictor buffer	Cycle	16 cycle (without memory)
Cache			Interleaving 4 controllers per node
L1 inst.	Perfect	Network	Pipelined point-to-point
L1 data	32 Kbyte, 4 way assoc., write-back 2 ports, 1 cycle, 16 outstanding misses	Network width	8 bytes/flit
L2	1 Mbyte, 4 way associative, write-back 10 cycle access, 16 outstanding misses	Switch buffer size	64 flits
L1/L2 bus Runs at processor clock			Switch latency 4 cycles/flit + 4 arbit.
Line size	64 bytes		
Code	Ocean Barnes Volrend Water-Nsq	Water-Spt	Radiosity FMM Raytrace
Input	258x258 16Kpartic. head 512 molec.	512 molec.	test 16K partic. Car 256

3.1 Execution Time Breakdown

We use the mean of the parallel phase execution time across all processors as the main metric to analyze performance improvements. The execution time of each application is split into four categories according to the system components causing each cycle loss. Instructions are assigned to the *Lock* (acquiring and releasing critical sections) and *Barrier* categories by using software marks. In the remaining code (not enclosed between the software marks), attributing stall cycles to specific components is complex in multiprocessor systems with out-of-order processors, because many events happen simultaneously. The algorithm used to categorize the remaining execution time is taken from the work of Pai et al. in [2]. The employed categories are *Compute* and *Memory*. Of course, the *Memory* category includes shared variable accesses both inside and outside critical sections.

3.2 Benchmarks

Table II shows the SPLASH-2 benchmarks [14] we tested. The applications have been compiled with a Test-and-Test&Set based synchronization library implemented with a Read-Modify-Write (*ldstub* in sparc) type operation. Every lock variable does not share its cache line with another lock, even in array locks. This has been achieved employing padding. Barriers are implemented with a binary tree. Radiosity and Volrend can be considered non-deterministic because different execution paths of the

tasks can produce different number of iterations on an active waiting loop and, as a consequence, the execution time experiences minor variations.

We simulate completely the parallel phase of each benchmark. In Figure 2 we show the execution time breakdown of the *base* experiment that will be used later to compare with the optimized options. As can be seen the Lock time is noticeably high, ranging from 1 % in FMM to 50 % in Radiosity.

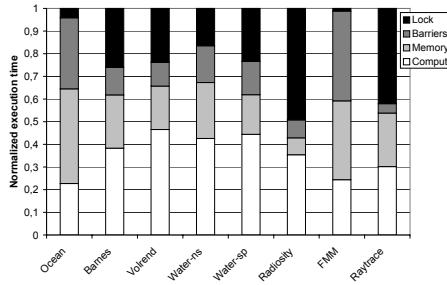


Fig. 2. Parallel execution time of the SPLASH-2 benchmarks

4 Critical Section Classification in SPLASH-2

In this section we analyze the code of each critical section of the SPLASH-2 benchmarks and we classify them in the types stated in Section 2.

As can be seen in Table 2, type A accumulates the highest number of critical sections (21) while Type B has only 7 members. The simplest A.0 class is the most populated and covers all the applications, followed in importance by A.1. Regarding type B, it is worth noting that subtypes B.0 and B.1 only appear each in three out of the eight applications.

Table 2. Critical section classification of the SPLASH-2 benchmarks. *Count* in Volrend protects two different codes in different execution phases, one is A.0 and the other A.1.

Benchmark	type A.0 Locks	type A.1 Locks	type B.0 Locks	type B.1 Locks
	SHVar += value	Several variables (e.g. linked lists)	Task queues	Complex data structures
Ocean	Id, Psiai, Psibi, Error			
Barnes	Count	Count		Cell[0:2047]
Volrend	Index, Count, QLock[32]		QLock[0:31]	
Water-ns	Index, IntrafVir, InterfVir, KinetiSum	PotengSum		Mol[0:511]
Water-sp	Index, IntrafVir, InterfVir, KinetiSum	PotengSum		
		SHLocks(0:700), free_patch, free_interaction, free_edge, free_element, free_elemvertex, avg_radiosity	tq[0:31].flock	
Radiosity	Index, Pbar, TaskCounter		tq[0:31].qlock	
FMM	Count	Mal		LockArray[0:1235]
Raytrace	Pid	Mem	WP[0:31]	

5 Experimental Results

In this section we first characterize each application according to lock contention and degree of data sharing. Second, we identify the key critical sections according to their influence in the overall execution time. Third, for every key critical section we analyze contention, sharing patterns and Latency. Fourth, we focus on false sharing removing and opportunities for collocation and other code optimizations. Finally, we show a sensitivity analysis of the optimized code performance with respect to hardware enhancements.

Critical section characterization. Figure 3 (a) plots data sharing (y-axis) and contention (x-axis) for each application, placing each critical section in only one of the four regions. A critical section is placed in the *No Data Sharing* half if its dominant sharing pattern is no-sharing. On the other hand, the critical section is placed on the *Contention* half if more than 5 % of its instances execute with 2 or more processors trying to enter the critical section. Then for each application we group all the critical sections of a given region in a ball, weighting its area according to the aggregated number of execution instances¹. The Lock time of all the critical sections included in a ball, with respect to the parallel execution time, is plotted close to each ball.

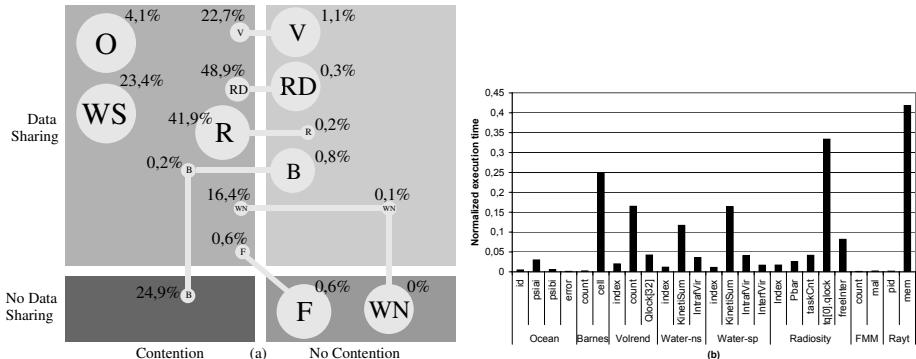


Fig. 3. (a) SPLASH-2 characterization based on contention and data sharing of their critical sections. (O:Ocean, WS:Water-sp, WN:Water-ns, B:Barnes, V:Volrend, RD:Radiosity, R:Raytrace, F:FMM). (b) Lock time of key critical sections.

Lock time. To identify the key critical sections we analyze the contribution of each critical section to the parallel execution time, selecting the most relevant. Figure 3 (b) plots the Lock time of relevant critical sections normalized to the parallel execution time. In several applications one single critical section has a

¹ The ball area is proportional to the sum of the number of executions of all the critical sections grouped in the ball, relative to the number of executions of all the critical sections of the application.

significant impact on the execution time due to its high contention (eg. `tq[0].qlock` in Radiosity and `Mem` in Raytrace).

Contention. Figure 4 (a) shows the average number of waiting processors for the key critical sections. We see, on average, between 7 and 10 processors waiting on each execution. A critical section can show a highly contended behavior whereas its contribution to the parallel execution time may be low. For example, three critical sections in Ocean have high contention (14 processors on average) but their impact on the parallel execution time is low because they execute only a few times, see Figure 3 (b).

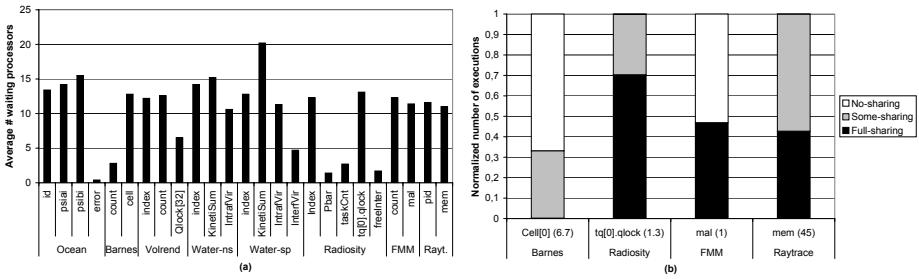


Fig. 4. (a) Average number of processors waiting on each key critical section execution. (b) Degree of data sharing of key critical sections not belonging to A.0 type; the average number of referenced cache lines appears in brackets.

Sharing patterns. In Figure 4 (b) we show the normalized number of occurrences of each sharing pattern for the key critical sections that are not A.0 type. The average number of referenced cache lines on each critical section execution appears in brackets. Of course, critical sections of A.0 type show only full-sharing. The locks `tq[0].qlock` (Radiosity) and `Mem` (Raytrace) show high degree of data sharing. The lock `Cell` in Barnes has a very low degree of data sharing and, finally, `Mal` in FMM shows equal amounts of full-sharing and no-sharing.

Latency. Finally, in Figure 5 (a) we show the average Latency, in execution cycles, of the key critical sections. In each bar we show compute and memory time. We see that in all the critical sections except two, the Latency almost equals the memory component. Note that the numbers are quite high, for instance, roughly half the critical sections spend 1000 cycles or more accessing memory. Therefore, reducing the memory component clearly appears as the main target of any optimization we are able to apply. Remember that the Lock time is proportional to the Latency times the number of contending processors, and so small Latency reductions can have a large impact on the Lock time.

5.1 Data Layout and Code Optimizations

First we deal with removing inefficiencies, detecting false sharing and applying other code optimizations. Next, we apply collocation, verifying that it achieves a positive effect.

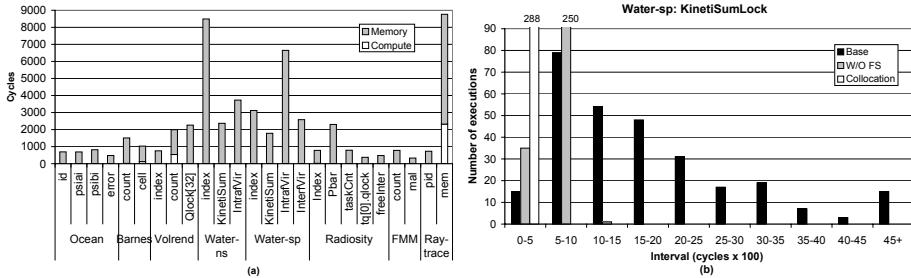


Fig. 5. (a) Average Latency of key critical sections. (b) Number of executions of the critical section *KinetiSum* (Water-sp) in each Latency interval for the Base experiment (*Base*), after removing false sharing (*W/O FS*) and after applying Collocation (*Collocation*).

When applying these code optimizations some changes in the Barrier time measured in Figure 2 can be observed. Due to optimization, the Barrier time can increase if some of the processors, but not the last one, arrive earlier to the barrier, but it can decrease if all the processors arrive earlier to the barrier. In Radiosity and Volrend there is an active waiting loop just before the barrier, therefore when the code is optimized the number of iterations will change and so will all the Barrier, Compute and Memory times.

False sharing. Comparing Figure 5 (a) with the classification in Table 2 we find some critical sections with an unusual behavior. For example in Water-sp the critical section *KinetiSum* has an average Latency of 1781 cycles but it is of A.0 type and does not have any conditional statement. Note that it is a too high Latency for a critical section that only reads and writes a single variable once (typical in type A.0 critical sections). Actually, if we add the raw latencies of whatever two memory accesses (276 and 377 cycles in average) and the negligible compute time, the resulting time is clearly lower than the measured Latency. In Figure 5 (b) we perform a more detailed analysis of this critical section by plotting the number of critical section executions (bar height) whose Latency is in a given interval (x-axis category). In the base experiment of this figure we see that there are significant variation in the measured Latency.

Analyzing the code of *KinetiSum* we observe the existence of false sharing [16]. Figure 5 (b) also shows the *KinetiSum* Latency once the false sharing has been removed through data padding (experiment W/O FS). We can see now that the Latency of almost all the executions is below 1000 cycles. Using this kind of Latency analysis the false sharing behavior is also detected in Ocean, Raytrace and Water-ns. The execution time without false sharing is shown in Figure 6 (a) (experiment W/O FS).

Optimizations in the code. In Barnes there is one specific lock belonging to the lock array Cell (type B.1) with high Lock time (see Figure 3 (b)), high Latency ($\simeq 1000$ cycles, see Figure 5 (a)) and very low data sharing (see Figure 4 (b)). A B.1 type contended lock showing a low degree of data sharing indicates

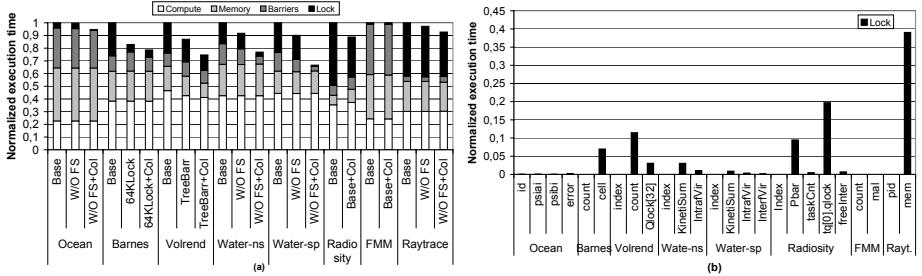


Fig. 6. (a) Normalized execution time for the base experiment (*Base*) after removing false sharing (*W/O FS*), increasing the size of the lock array in Barnes (*64KLock*), replacing a lock-based barrier with a tree barrier in Volrend (*TreeBarr*), and applying Collocation (*Col*). (b) Lock time of key critical sections after applying all optimizations.

a too much coarse-grained protection of the shared data. Therefore, we have increased the lock array Cell from 2048 to 65536 locks, reducing 77 % Barnes lock time (Figure 6 (a), experiment *64KLock*).

In the Volrend benchmark the lock Count has high Lock time and Latency (see Figures 3 (b) and 5 (a)), but it does not suffer from false sharing. Analyzing the code we see that Count is of type A.0 but it implements a barrier by means of a shared counter. We replaced it with a simple binary tree barrier and results are shown in Figure 6 (a), experiment *TreeBarr*. All components in the execution time are reduced except the barrier component because the time of the new tree barrier is accumulated to the barrier category. The other categories decrease because one lock, the corresponding critical section and its control loop are eliminated.

Collocation. When several variables stored in the same line are used by different processors performance degrades (false sharing) but, on the other hand, if these variables are used by the same processor performance can improve. The software technique named Collocation aims at decreasing the latencies associated with data accesses by placing the lock and the variables together in the same line [5]. Bar *Collocation* in Figure 5 (b) shows the KinetiSum Latency distribution when Collocation is applied. We can see that all the KinetiSum executions have a Latency below 500 cycles (with an average Latency of 25 cycles). This is because the read and write accesses to the protected variable now almost always hit in the cache.

The bars *+Col* in Figure 6 (a) show the execution time breakdown when Collocation is applied on top of the previous optimizations. As can be seen, collocation decreases execution time further on all applications, between 5.2 % in Ocean and 23.0 % in Water-sp, except in FMM. In order to complete the picture, Figure 6 (b) shows the Lock time of the key critical sections. As we can see by comparing it with Figure 3 (b) the Lock time has been eliminated or significantly reduced in most key critical sections.

Summarizing, after applying all the optimizations suggested in our methodology, the Baseline system execution time is reduced between 5.5 % in Ocean and 33.4 % in Water-sp. However, there is no reduction in FMM, which has no Lock time and therefore is insensitive to these kinds of optimizations. A significant lock time remains in Radiosity and Raytrace mostly, which would require to restructure two key critical sections, namely *tq[0].qlock* in Radiosity and *Mem* in Raytrace.

Sensitivity of code optimizations. In this section we evaluate the sensitivity of code optimizations to architectural enhancements. First we simulate processors that prefetch on the memory operations blocked by the memory consistency constraints, and that perform store buffering [7]. Later we simulate a directory cache in the coherence controller (four-entry fully associative) whose entries hold both the data line and its directory information, and are looked up before in the directory [15,8,9]. A line is placed in the directory cache only when it is requested by a processor and it is in Exclusive state in another processor node.

Figure 7 shows execution time of the enhanced processors normalized to the base experiment, running both the original and the optimized codes, including the base experiment for comparison. As can be seen, the enhanced processors run faster, except with the directory cache in Raytrace. In Raytrace, within the critical section *Mem*, the number of accessed and updated lines is high which causes a lot of capacity misses in the directory cache. Anyway, and more importantly, the optimized code on the enhanced processors always executes in less time, excluding FMM which is insensitive to optimization.

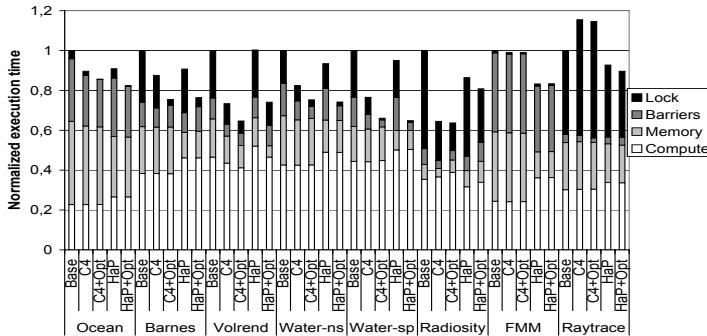


Fig. 7. Execution time normalized to the base experiment of several architectural enhancements with and without optimizations: four-entry cache in the coherence controller (*C4*, and *C4+Opt*), prefetch and store buffering (*Hap* and *Hap+Opt*)

6 Related Work

Contemporary processors provide hardware counters that return a count of either cycles or performance-related events. Performance analysis tools insert library calls in the program to query the hardware counters during program execution.

These tools provide graphical interface and correlate performance metrics with the program source code [16][17]. Instead, we use a simulator allowing to measure the overheads at the instruction level and to eliminate the nondeterminism introduced by real hardware. Therefore, we can take fine grain statistics, analyze data address streams or break down the execution time of critical sections accurately, which is difficult to accomplish in real systems. Also, we are able to connect and disconnect architectural optimizations that might introduce noise in the measures used in a first-step analysis.

Other performance analysis tools instrument the source program in order to obtain statistics or address traces but, as a consequence, the data address space can be disturbed [3][18]. The disruption may affect the absolute position of the data elements, which in turn may affect the cache behavior. Moreover, tools that use trace driven simulation require to implement a given interleave of the memory reference streams [3]. Although we also use hooks to instrument the code, the simulator allows us to eliminate all bookkeeping operations introduced by instrumentation. Besides, the execution-driven simulation already interleaves the memory reference streams in a straightforward way.

7 Concluding Remarks

Critical sections are used in parallel applications to ensure serialized access to shared data structures, leading to a potential performance bottleneck. In this paper we propose a methodology to characterize and classify critical sections in order to guide optimizations in DSM multiprocessors. To characterize a critical section we use three parameters measured with an execution driven simulator that allows to take fine grain statistics: a) lock contention, b) degree of data sharing in consecutive executions of the same critical section, and c) Latency. The fine grain statistics taken in the simulation framework are correlated with a static classification. The static classification takes into account how locks are associated with their protected data.

Correlating the static classification of each critical section with its dynamic characterization, inefficiencies may be detected, isolated and understood. Characterization is used to guide optimization opportunities and classification allows us to identify the best suited optimization for each application. Data layout optimizations in critical sections have been introduced to reduce the execution time. We show that, for a highly contended short critical section with high degree of data sharing, data layout optimizations not only reduce the critical section Latency but also, and to a large extent, the Lock time, significantly reducing the execution time. Moreover, we show that the proposed optimizations hold their effectiveness even when considering architectural enhancements in the processor and the coherence controller.

In this work we have optimized the original version of SPLASH-2 for a DSM environment by carefully tuning the lock-based synchronization performance. In a future work we plan to test the effectiveness of the proposed optimizations in a design with very different architectural tradeoffs, such as for example, a chip multiprocessor.

References

1. Fernández, R., García, J.: rsim x86:a cost-effective performance simulator. In: Proc. 19th European Conference on Modelling and Simulation ECMS (2005)
2. Pai, V., Ranganathan, P., Adve, S.: rsim reference manual version 1.0. Technical report 9705, Dept. Electrical and Computer Eng., Rice University (1997)
3. Marathe, J., Mueller, F.: Source-code-correlated cache coherence characterization of OpenMP benchmarks. *IEEE Transactions on Parallel and Distributed Systems* 18(6), 818–834 (2007)
4. Eggers, S.J., Jeremiassen, T.: Eliminating false sharing. In: Proc. Int. Conf. Parallel Processing, vol. I, pp. 377–381 (1991)
5. Kagi, A., Burger, D., Goodman, J.: Efficient synchronization: let them eat QOLB. In: Proc. 24th ISCA, pp. 170–180 (1997)
6. Torrellas, J., Lam, M., Hennessy, J.: False sharing and spatial locality in multiprocessor caches. *IEEE Trans. Computers* 43(6), 651–663 (1994)
7. Gharachorloo, K., Gupta, A., Hennessy, J.: Two techniques to enhance the performance of memory consistency models. In: Proc. ICPP, pp. 355–364 (1991)
8. Michael, M., Nanda, A.: Design and performance of directory caches for scalable shared memory multiprocessors. In: Proc. 5th HPCA (1999)
9. Woodacre, M., Robb, D., Roe, D., Feind, K.: The SGI altix 3000 global shared-memory architecture. White paper silicon graphics inc., SGI (2003)
10. Anderson, T.: The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel and Distrib. Systems* 1(1), 6–16 (1990)
11. Graunke, G., Thakkar, S.: Synchronization algorithms for shared memory multiprocessors. *IEEE Computer* 23(6), 60–69 (1990)
12. Mellor-Crummey, J., Scott, M.: Algorithms for scalable synchronization on shared memory multiprocessors. *ACM Trans. Computer Systems* 9(1), 21–65 (1991)
13. Laudon, J., Lenoski, D.: The sgi origin: A cc-NUMA highly scalable server. In: Proc. 24th ISCA (1997)
14. Woo, S., et al.: The SPLASH-2 programs: Characterization and methodological considerations. In: Proc. 22th ISCA, pp. 24–36 (1995)
15. Acacio, M., González, J., García, J., Duato, J.: Owner prediction for accelerating cache-to-cache transfer misses in a cc-NUMA architecture. In: Proc. 16th Int. Conf. on Supercomputing (2002)
16. Browne, S., Dongarra, J., Garner, N., London, K., Mucci, P.: A scalable cross-platform infrastructure for application performance tuning using hardware counters. In: ACM/IEEE Supercomputing Conference, p. 42 (2000)
17. De Rose, L., Reed, D.: Svpablo: A multi-language architecture-independent performance analysis system. In: Int. Conf. Parallel Processing, pp. 311–318 (1999)
18. Mellor-Crummey, J., Fowler, R., Whalley, D.: Tools for application-oriented performance tuning. In: Proc. 15th Int. Conf. Supercomput, pp. 154–165 (2001)

Topic 3

Scheduling and Load Balancing

Introduction

Emmanuel Jeannot*, Ramin Yahyapour*, Daniel Grosu*, and Helen Karatza*

Scheduling and load balancing are key components of any resource management system. They are responsible for making the best use of available resources by planning, allocating and redistributing computational tasks onto these resources. Typical optimization criteria are the minimization of execution times and response times. However, additional criteria like power consumption, quality-of-service requirements or even multi-criteria problems have recently been investigated.

Scheduling and load balancing have been active areas of research throughout decades. These techniques are crucial for implementing efficient parallel and distributed applications. As parallel and distributed systems are evolving rapidly, new scheduling problems appear and novel solutions are sought which can be applied to these scenarios.

In short, the goal of this topic is to serve as a forum where efficient and robust scheduling, load-balancing, and/or resource management algorithms are proposed for modern parallel and distributed systems such as clusters, grids, and global computing platforms.

The selection process for this topic area was highly competitive. A total of 29 papers have been submitted. All papers were reviewed by at least three independent reviewers. Finally, only 9 papers have been accepted which addressed new and exciting challenges and presented innovative solutions. The accepted papers range from theory to practice and address various problems such as energy-aware scheduling, dynamic load balancing for linear algebra, steady-state scheduling, dynamic scheduling for heterogeneous systems, the use of genetic algorithms, fault-tolerance, packet reordering, task or replica migration, and replica placement.

Finally, we would like to thank all the reviewers, for their time and effort, who helped us in the selection process.

* Topic Chairs.

Dynamic Load Balancing of Matrix-Vector Multiplications on Roadrunner Compute Nodes

José Carlos Sancho and Darren J. Kerbyson

Performance and Architecture Laboratory (PAL),
Los Alamos National Laboratory, NM 87545, USA
`{jcsancho,djk}@lanl.gov`

Abstract. Hybrid architectures that combine general purpose processors with accelerators are currently being adopted in several large-scale systems such as the petaflop Roadrunner supercomputer at Los Alamos. In this system, dual-core Opteron host processors are tightly coupled with PowerXCell 8i accelerator processors within each compute node. In this kind of hybrid architecture, an accelerated mode of operation is typically used to off-load performance hotspots in the computation to the accelerators. In this paper we explore the suitability of a variant of this acceleration mode in which the performance hotspots are actually shared between the host and the accelerators. To achieve this we have designed a new load balancing algorithm, which is optimized for the Roadrunner compute nodes, to dynamically distribute computation and associated data between the host and the accelerators at runtime. Results are presented using this approach, for sparse and dense matrix-vector multiplications, that show load-balancing can improve performance by up to 24% over solely using the accelerators.

1 Introduction

The unprecedent need for power efficiency has primarily driven the current design of hybrid computer architectures that combine traditional general purpose processors with specialized high-performance accelerators. Such a hybrid architecture has been recently installed at Los Alamos National Laboratory in the form of the Roadrunner supercomputer [1]. This system was the first to achieve a sustained performance of over 1 *PetaFlop/s* on the LINPACK benchmark.

In Roadrunner, dual-core Opteron host processors are tightly coupled with PowerXCell 8i processors [2]—an implementation of the Cell Broadband-Engine architecture (Cell BE) with high double-precision floating-point performance—within each compute node. This hybrid architecture can support several types of processing modes including: host-centric, accelerator-centric, and an accelerated mode of operation. The characteristics of an application determines which mode is most suitable. The host-centric mode can be thought of as the traditional mode of operation where applications solely use the host Opteron processors. In the accelerator-centric mode applications solely run on the PowerXCell 8i, an example that follows this mode can be found in the application VPIC [3],

Gordon Bell Prize finalist at SC08. In the accelerated mode, both the Opteron and PowerXCell 8i are used in such a way that performance-critical sections of computation are off-loaded to the PowerXCell 8i accelerators leaving the rest of the code to run on the host Opterons. SPaSM, a molecular dynamics code, is an example of an application that followed this accelerated approach [4], also Gordon Bell Prize finalist at SC08.

A variant of the accelerated mode is to share the performance hotspots between both the accelerator and host processors for simultaneous processing. The benefit of this is a potential gain in performance, since the computation power of both the host processors and accelerators can be harnessed simultaneously. The computation power of the host processors may be orders of magnitude smaller than that of the accelerators but at large-scale, including Roadrunner, the performance gain can be significant and thus should be exploited. However, this kind of accelerated mode increases complexity - extra tools are required in order to efficiently and dynamically load balance between the hosts and accelerators at runtime. Undertaking such a load balance during application execution is desirable in this context as it is difficult to determine costs associated with individual computations at compile-time, and also there may be changes in the amount of data to compute per processor during runtime which can result in repartitioning across nodes.

This paper addresses this challenge and presents a load-balancing algorithm in order to dynamically distribute the computation and associated data between the host Opterons and the PowerXCell 8i accelerators at runtime in the compute nodes of Roadrunner. For illustration purposes we address the common operation of matrix-vector multiplications on the form of $y_{i+1} = y_i + Ax$, where A is either a sparse or dense matrix and x and y are dense vectors. These operations are commonly found in scientific applications and are prime candidates to offload to accelerators. Results show that the dynamic load-balancing algorithm can improve the performance of these operations by up to 24% when using both host and accelerator processors in comparison to solely using the accelerators. In addition, the determination of the optimal load balance converges quickly taking only 7 iterations. Although the results as presented consider a 4-process parallel job (one compute node of Roadrunner), there is nothing to prevent our technique to being applied to larger process counts up to a system-wide parallel job because the scope of our technique is at the process level rather than at the system level.

The rest of this paper is organized as follows. Section 2 describes the architecture of a Roadrunner compute node. Section 3 describes our load-balancing algorithm. Section 4 includes experimental results from a Roadrunner node. Section 5 summarizes related work on matrix vector multiplications on hybrid architectures. And finally, conclusions from this work are given in Section 6.

2 The Roadrunner Compute Node

A compute node of Roadrunner is built using three compute blades (one IBM LS21 blade and two IBM QS22 blades) and one interconnect blade as shown in Figure 1. Each IBM LS21 blade contains two 1.8GHz dual-core Opteron

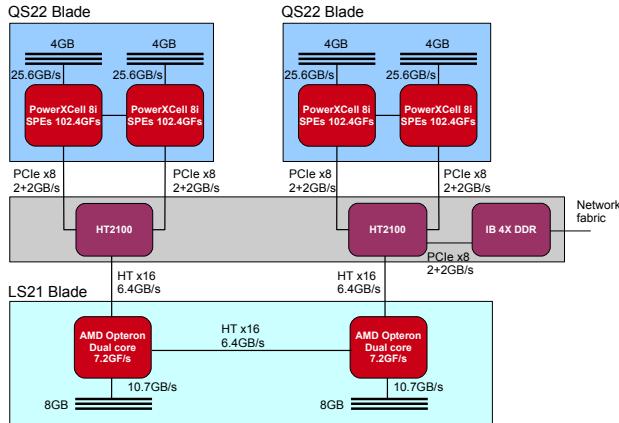


Fig. 1. The structure of a Roadrunner compute node

processors, and a single IBM QS22 blade contains two 3.2GHz PowerXCell 8i processors [2]. The fourth blade interconnects the three compute blades using two Broadcom HT2100 I/O controllers. These controllers convert the Hyper-Transport x16 connections from the Opterons to PCIe x8 buses — one to each PowerXCell 8i. In this configuration each Opteron core is uniquely paired with a different PowerXCell 8i processor when using the accelerated mode of operation.

The PowerXCell 8i processors have approximately 95% of the peak floating-point performance and 80% of the peak memory bandwidth of a node. Each PowerXCell 8i consists of eight Synergistic Processing Elements (SPEs) and one Power Processing Element (PPE). The eight SPEs have an aggregate peak performance of 102.4 GFlops/s (double-precision), or 204.8 Flops/s (single-precision) whereas dual-core Opteron has a peak performance of 7.2 GFlops/s (double-precision) or 14.4GFlops/s (single-precision). Therefore, a single PowerXCell 8i can potentially accelerate a compute-bound code by up to $28 \times (102.4/3.6)$ over a single Opteron core. In addition, each PowerXCell 8i processor has substantially more memory bandwidth than the Opterons, 25.6 GB/s compared to 10.7GB/s for a dual-core Opteron.

The PPE is a PowerPC processor core which runs a linux operating system (one per blade), and manages the SPEs. The SPEs are in-order execution processors with a two-way SIMD operation that do not have a cache. Instead they can directly access a 256KB high-speed memory called a *local store* which is explicitly accessed by direct memory access (DMA) transfers from the PPE memory space. Each compute node has a total of 32GB of memory, 8GB for each Opteron and 4GB for each PowerXCell 8i.

3 The Dynamic Load Balance Algorithm

In this section we describe our dynamic load-balancing algorithm applied to matrix-vector multiplication. These operations can be very time-consuming in

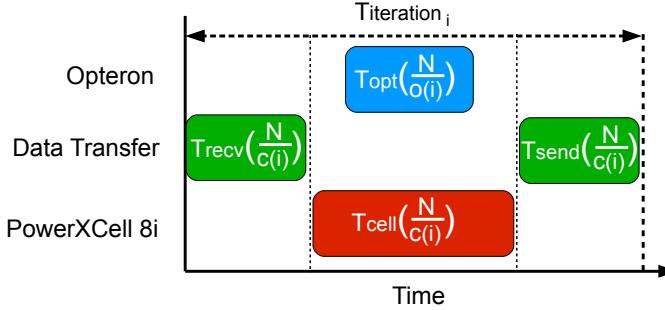


Fig. 2. Breakdown of iteration time (non-overlapping transfers)

codes such as iterative solvers where the matrix-vector multiplication, $y_{i+1} = y_i + Ax$, is performed once or more in each iteration of the application. We chose a single row of the matrix A as the smallest granularity of load balancing the data between the Opteron and PowerXCell 8i. The goal of the load-balancing algorithm is to find an optimal partitioning of the matrix rows to minimize the runtime when this calculation is performed multiple times as in iterative solvers. Formally, the load-balancing problem can be described as to minimize the following expression,

$$\max_{i=1}^n \left(T_{opt}\left(\frac{N}{o(i)}\right) + T_{trans}\left(\frac{N}{c(i)}\right), T_{cell}\left(\frac{N}{c(i)}\right) + T_{trans}\left(\frac{N}{c(i)}\right) \right) + T_{house}\left(\frac{N}{c(i)}\right)$$

when n is the number of times that the matrix-vector multiplication is performed; N is the number of matrix rows; $o(i)$ and $c(i)$ are the ratios in the amount of rows assigned to the Opteron and for the PowerXCell 8i, respectively, so $\frac{1}{o(i)} + \frac{1}{c(i)} = 1$; $T_{opt}(x)$ and $T_{cell}(x)$ are the times to perform the corresponding matrix-vector multiplications on x rows on the Opteron and PowerXCell 8i, respectively; $T_{trans}(x)$ is the sum of times for receiving data to compute on the PowerXCell 8i ($T_{recv}(x)$) and for sending back the results to the Opteron ($T_{send}(x)$); and finally, $T_{house}(x)$ is the *housekeeping* time associated with the execution of the load-balancing algorithm (described below) and formatting data for processing on the PowerXCell 8i. This formatting involves setting up of the various DMA transfers in order to iteratively transfer data in/out to/from SPEs and the replication of data structures in main memory in order to enforce the alignment of the DMA transfers. Therefore, the goal of the optimization problem is to dynamically define the function $o(i)$, $\forall i, 1 \leq i \leq n$ that minimizes the above expression; and hence, function $c(i)$ will be defined as $\frac{1}{1 - \frac{1}{o(i)}}$.

We follow the operation of iterative solvers where the data that is transferred in and out of the operation in each iteration are the vectors y_i and y_{i+1} respectively. The matrix A is considered constant as in most iterative solvers, and hence it does not need to be transferred to the PowerXCell 8i each iteration. Similarly, the vector x also does not need to be transferred each iteration as it is computed internally based on the residuals.

The optimization problem can be simplified to the problem of minimizing $\max\left(T_{opt}\left(\frac{N}{o(i)}\right) + T_{trans}\left(\frac{N}{c(i)}\right), T_{cell}\left(\frac{N}{c(i)}\right) + T_{trans}\left(\frac{N}{c(i)}\right)\right)$ when the number of iterations is large enough that $T_{house}\left(\frac{N}{c(i)}\right)$ is negligible. This case is illustrated in Figure 2 that shows the elapsed times on the Opteron, PowerXCell 8i, and the intranode-connection network in the particular case when data transfers are not overlapped with computation. Therefore, we want to minimize both $T_{opt}\left(\frac{N}{o(i)}\right) + T_{trans}\left(\frac{N}{c(i)}\right)$ and $T_{cell}\left(\frac{N}{c(i)}\right) + T_{trans}\left(\frac{N}{c(i)}\right)$ at the same time. By distributing the data carefully between processors (defining the function $o(x)$) it is possible to achieve the optimal balance that minimizes the above expression. For example, in the case that the PowerXCell 8i has too much to compute, we can move some data to the Opteron which reduces both $T_{cell}\left(\frac{N}{c(i)}\right)$ and $T_{trans}\left(\frac{N}{c(i)}\right)$ at expense of increasing $T_{opt}\left(\frac{N}{o(i)}\right)$. Careful attention should be taken to prevent the case that the Opteron has too much data to compute, $T_{opt}\left(\frac{N}{o(i)}\right) > T_{cell}\left(\frac{N}{c(i)}\right)$, which will also increase the iteration time. In the converse case, that the Opteron has too much data, some data can be moved from the Opteron to the PowerXCell 8i. Note again that assigning more data to the PowerXCell 8i in the next iteration, $i + 1$, means that both the $T_{cell}\left(\frac{N}{c(i+1)}\right)$ and $T_{trans}\left(\frac{N}{c(i+1)}\right)$ will be increased. And therefore, the iteration time may be larger because the $T_{trans}\left(\frac{N}{c(i+1)}\right)$ might be too high to offset the reduction in time on the Opteron, $T_{trans}\left(\frac{N}{c(i+1)}\right) - T_{trans}\left(\frac{N}{c(i)}\right) > T_{opt}\left(\frac{N}{o(i)}\right) - T_{opt}\left(\frac{N}{o(i+1)}\right)$. It can also occur that the cell has to much data to compute with respect to the Opteron, $T_{cell}\left(\frac{N}{c(i+1)}\right) > T_{opt}\left(\frac{N}{o(i+1)}\right)$.

On the other hand, when data transfers can be fully overlapped with computation, the load balancing is simplified to the case of making the compute-times on both the Opteron and PowerXCell 8i equal, $T_{opt}\left(\frac{N}{o(i)}\right) \simeq T_{cell}\left(\frac{N}{c(i)}\right)$, in order to minimize the following expression $\max(T_{opt}\left(\frac{N}{o(i)}\right), T_{cell}\left(\frac{N}{c(i)}\right))$. This is an ideal case that might be difficult to achieve in a real scenario because it depends on the application's data dependencies— data is not available yet because it needs to be combined with other data such as in the case of iterative solvers—, and the support of asynchronous operations on the communication system. Hence, the common scenario is that communications are only partially overlapped and the optimization problem described in Figure 2 applies.

The load-balancing algorithm proposed is based on combining the following three basic approaches: **accelerator-centric**, **performance-based**, and **trial-and-error** in order to converge at the optimal state as quickly as possible. This algorithm is comprised of five states as depicted in Figure 3. In the first state, we take an **accelerator-centric** approach where $o(1)$ is initialized to be $\frac{Peak_{cell}}{Peak_{opt}}$, where $Peak_{opt}$ and $Peak_{cell}$ are the peak flop performance of the PowerXCell 8i and Opteron, respectively. In Roadrunner, this is initialized to be 28, see Section 2. We use the peak performance of the processors as an starting point as this is available a priori. In principle, we do not know anything about the characteristics of the code and the peak flop performance is a safe alternative in

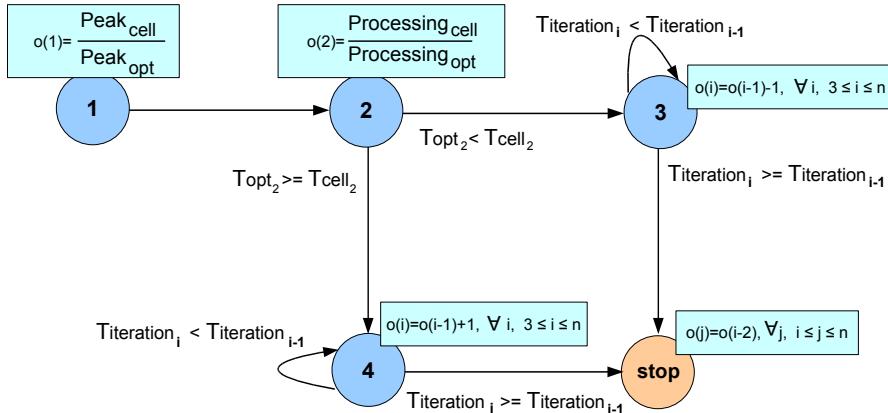


Fig. 3. States of the load-balance algorithm for n iterations

this architecture with respect to the peak memory bandwidth because most of the work will be performed on the PowerXCell 8i rather than the Opteron.

In the second state, we take a **performance-based** approach since we can collect actual timing information from the previous iteration. The principle of a performance-based approach is to distribute data based on how well the different processors perform, and thus allowing the algorithm to quickly converge to the optimal ratio. This is achieved by collecting the times, T_{opt} and T_{cell} , in order to calculate the processing rates, $Processing_{opt}$ and $Processing_{cell}$, for both the Opteron and PowerXCell 8i, thus $o(2) = \frac{Processing_{cell}}{Processing_{opt}}$. Note that T_{cell} and T_{trans} are measured independently instead of combining them into a single metric. This distinction is more efficient than the typical combination approach of as will be shown in the next section.

Finally, the third and fourth state are performed using a **trial-and-error** load-balancing strategy until the optimal balance is achieved. This is done by carefully assigning more or less data on the Opteron in order to not increase the $T_{iteration}$ for the next iteration. In particular, the third state is reached from the second state when the Opterons have not enough data to compute, case of $T_{opt_2} < T_{cell_2}$; and the forth state is reached from also the second state when the Opterons have too much data to compute, case of $T_{opt_2} \geq T_{cell_2}$. Note that these additional steps are not included in a typical performance-based load-balancing strategy, but they were necessary for the case of this particular architecture. In particular, the third state gradually decreases $o(i)$ by one assigning more data to the Opteron. Similarly, the fourth state gradually increases $o(i)$ by one assigning less data to the Opteron. Convergence is achieved when the current $T_{iteration_i}$ is higher than the previous $T_{iteration_{i-1}}$ time stopping the algorithm in state *stop*. In this state, we set up $o(j)$, $\forall j, 1 \leq j \leq n$ to the value used two iterations previously, $o(i-2)$. Note again, that for the case of fully overlapping transfers these additional states might not lead to the optimal balance as the second state should already give a good balance due to the fact that it is based on the

Table 1. Description of the matrices used in the evaluation

Name	Dimensions	Non-zeros	Description
Dense matrix	$2K \times 2K$	4M	Regular dense matrix
Sparse Harbor	$47K \times 47K$	2.37M	3D CFD of Charleston harbor
Sparse Dense	$2K \times 2K$	4M	Matrix in sparse format
Sparse Fluid	$20.7K \times 20.7K$	1.41M	Fluid structure interaction turbulence
Sparse QCD	$49K \times 49K$	1.90M	Quark propagators (QCD/LGT)
Sparse Ship	$141K \times 141K$	3.98M	FEM ship section/detail production
Sparse Cantiveler	$62K \times 62K$	4M	FEM cantiveler
Sparse Spheres	$83K \times 83K$	6M	FEM concentric spheres

achieved processing rate and the transfer time does not impact on the iteration time. However, these states are necessary in the case of partially overlapping and non-overlapping transfers where the transfer time does actually impact the iteration time.

4 Evaluation

We evaluate our load-balancing technique on a Roadrunner compute node as described in Section 2. A four-process parallel job— one process per each of the four Opteron cores in the Roadrunner node— was executed in the accelerated mode of operation. Each process performed the same double-precision floating-point matrix-vector multiplication several times. At the end of the calculation all the processes synchronize in order to account for the worst time. Timing data presented below are averages over multiple runs. We use the DaCS communication library [5] for communicating between Opteron and PowerXCell 8i processors, and OpenMPI version 1.3b [6] message passing library for the synchronization across Opteron cores. The Cell BE SDK version 3.1 was used to compile the code for the PowerXCell 8i processors.

We evaluated the performance of our load-balance technique, *Optimized balance*, as well as for the case of using our load-balance algorithm but considering T_{trans} in combination with T_{cell} , *Balance transfer*. Also for comparison purposes we evaluated the performance of using no load balancing in two cases: using only Opterons and using only PowerXCell 8i processors. In addition, we show results for a *Greedy* strategy that searches for the optimal load balance by exploring a wide range of distributions: it starts with the default distribution ($o(1) = \frac{Peak_{cell}}{Peak_{opt}}$) and gradually decrements it in steps of one every iteration to when all work is performed by the Opterons. The experiments were conducted on a dense matrix and on seven sparse matrices from a wide variety of actual applications as listed in Table 1 where Sparse Dense is a dense matrix, but formatted in the sparse format. We used the *Compressed Storage Row* (CSR) format [7] for defining the sparse matrices.

4.1 Results

Figure 4 shows the iteration time for the *Greedy*, *Optimized balance*, and the *Balance transfer* techniques on the sparse matrix Harbor. As can be seen, the minimum execution time is found at iteration 24 for the *Greedy* technique, where $o(24) = 5$. At this point the optimal load balance is achieved and the execution time is improved by 15% with respect to using the performance-based ratio ($o(2) = \frac{\text{Processing}_{cell}}{\text{Processing}_{opt}}$), and $3.4 \times$ with respect to $o(28) = 1$ where all the work is performed by the Opterons. The *Greedy* technique can easily find the optimal balance, but at the expense of a longer converge time (28 iterations) which is undesirable. In contrast, the *Optimized balance* technique converges faster and is able to find the optimal balance after only 5 iterations for this matrix. Converging faster is desirable as there is extra overhead due to housekeeping per iteration which could be significant, see Section 3. In the case of the Roadrunner compute node this time is around 60ms per iteration (results not shown). For the case of the *Balance transfer* technique we can see that the load-balance algorithm does not converge to the optimal solution. This is because including the T_{trans} in the T_{cell} makes the performance-based ratio too low ($o(2) = 2$) for this architecture due to T_{trans} being high. This forces the load-balancing search to stop too early as the next ratio tried in state 3 of the algorithm, unfortunately, does not use the accelerators at all ($o(3) = 1$), and hence $T_{iteration}$ is higher than the previous one.

Figure 5 illustrates how the *Optimized balance* technique converges to the optimal balance during the first 5 iterations by showing the corresponding times T_{opt} , T_{cell} , T_{trans} , and $T_{iteration}$ for each iteration. On the first iteration, T_{opt} is too small compared with the T_{cell} because $o(1) = \frac{\text{Peak}_{cell}}{\text{Peak}_{opt}}$, $o(1) = 28$ yields too little work for the Opterons compared with the PowerXCell 8i. On the second

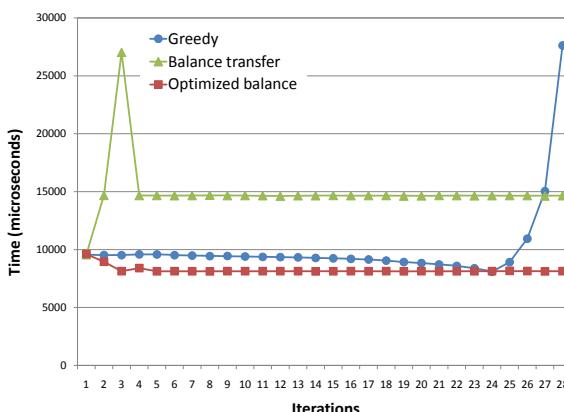


Fig. 4. Iteration time for the Greedy, Optimized balance, and Balance transfer techniques on the sparse matrix Harbor

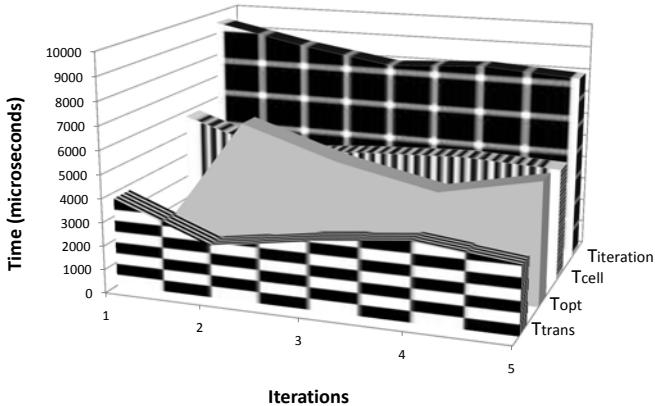


Fig. 5. Iteration time breakdown for the Optimized balance technique on the sparse matrix Harbor (first five iterations)

iteration, the ratio is already fixed to the current performance of the processors ($o(2) = \frac{Processing_{cell}}{Processing_{opt}}$, $o(2) = 4$), but actually results in too much work for the Opterons. On the third and fourth iterations, the load-balance algorithm is in state 4 gradually increasing the ratio ($o(3) = 5$, $o(4) = 6$) in order to reduce the work on the Opterons. During this, it is found that the third iteration results in a better $T_{iteration}$ time than the fourth iteration, and so the algorithm stops on the fifth iteration taking the best tested ratio, $o(3) = 5$, for subsequent iterations, $\forall i, 5 \leq i \leq n$. At the optimal balance, 65%, 60%, and 35% of the iteration time is spent on the T_{opt} , T_{cell} , and T_{trans} , respectively.

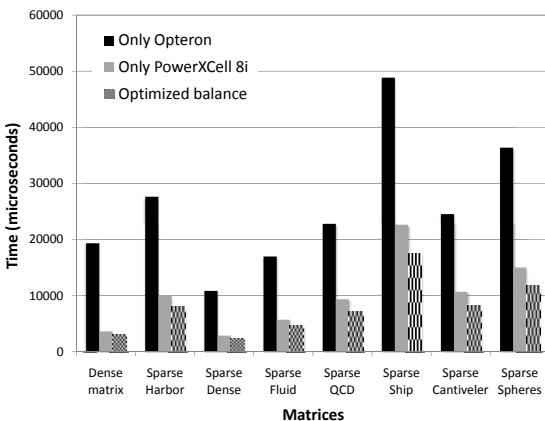


Fig. 6. Execution time for each matrix when using: only the Opteron, only the PowerXCell 8i, and the Optimized balancing technique

Figure 6 summarizes the execution iteration time for the suite of matrices evaluated when using *Optimized balance* (once the algorithm converged), when using the Opterons only and when using the PowerXCell 8is only. As can be seen, the *Optimized balance* achieves the best runtimes for all the matrices evaluated. In particular, for the dense matrix the performance improvement is 14% for the *Optimized balance* in comparison to using only the PowerXCell 8i. For the sparse matrices the improvements are 19%, 18%, 19%, 23%, 23%, 24%, 22% for the sparse matrices Harbor, Dense, Fluid, QCD, Ship, Cantiveler, and Spheres respectively. These improvements are mostly due to the fact that the computation of the sparse matrices is actually memory bound and thus take advantage of the relatively better memory performance of the Opterons rather than their flop performance. As expected, the improvements with respect to the Opteron are more noticeable, ranging from 4× on the sparse Fluid up to 6× for the dense matrix. Also, the number of iterations for the load-balancing algorithm to converge for these matrices is small—for the sparse matrices 5 iterations are required for convergence whereas the dense matrix required 7 iterations (results not shown).

5 Related Work

Matrix operations including sparse matrix-vector multiplications (SpMV) are key computational kernels in many scientific applications, and thus have been extensively studied. Today most work is focused on implementing these operations on emerging accelerator architectures including the Cell BE [8], FPGAs [9], and GPUs [10], as well as multi-core processors [8]. Although our SpMV implementation might not be so highly tuned for a particular processor in comparison to other implementations, they could be incorporated into our accelerator and host load-balancing method in order to improve overall performance.

On the other hand, there has been very little work on load-balancing matrix operations on hybrid (host-accelerator) architectures since typically they are fully offloaded to the accelerators. However, there is a significant work on load-balancing matrix operations like the SpMV on heterogeneous network of workstations (HNOWs) [11,12,13]. These systems are composed of non-uniform processors, network, and memory resources which partially resemble the hybrid platform studied in this work. For HNOWs most of the algorithms are optimized based on the characteristics of the target system. In fact as stated in [11] there is not a unique general solution for all platforms but rather different schemes are best for different applications and system configurations. This result is interesting because it suggests that there should be an efficient load balancing technique as well for our target platform. In particular, our platform is quite different from HNOWs. The processors are tightly attached to each other, so communications are much faster than in HNOWs. Also, there is a huge difference in the computing power of the processor types. These two features open new considerations in the design of load-balancing algorithms that they were not previously important. For example, in this new environment with fast communications it makes more sense to explore fine-grain load balancing algorithms, such as the one proposed in this paper, based on a trial-and-error strategies.

Additionally, in most of the load-balancing strategies for HNOWs distributing the load in proportion to the computing speed of the processors always leads to a perfectly balanced distribution [11,12]. However, we found that this strategy was not enough to achieve an optimal solution for the hybrid, host-accelerator architecture of Roadrunner.

Notwithstanding, it would be interesting to evaluate as future work the suitability of our proposed load-balancing algorithm to other hybrid platforms. In that regard, we could apply our dynamic load balancing technique into execution environments such as StarPU [14]—an unified execution model various accelerators—in order to dynamically determine the granularity of the tasks on different accelerators.

6 Conclusions

An optimized load-balancing algorithm has been presented in this paper to substantially increase the performance of a Roadrunner compute node. We have demonstrated that the proposed load-balance algorithm achieves a significant performance improvement, up to 24%, when simultaneously using both host (Opteron) and accelerator (PowerXCell 8i) processors in comparison to solely using the PowerXCell 8i processors in a traditional accelerated mode of operation. The load balancing was evaluated for matrix-vector multiplications which are commonly found in scientific applications.

These improvements come from the concurrent exploitation of the computation power of the host Opteron processors at the same time as the PowerXCell 8i accelerators for processing hotspot computations rather than uniquely offloading to the accelerators. These results suggest that the traditional accelerated mode of operation is not efficient enough to exploit the full potential of hybrid architectures including Roadrunner. With effective load-balancing techniques a more complex, but better accelerated mode of operation, can be enabled exploiting concurrently the full potential of all the available processors. In addition, the load-balance algorithm was carefully optimized to provide fast convergence time (7 iterations) making it sufficiently efficient to run during the execution of an application. This feature is desirable in order to dynamically adapt to the characteristics of the code, and thus it can potentially serve as a general load-balancing algorithm on this platform for other hotspot computations.

Acknowledgments

This work was funded in part by the Advanced Simulation and Computing program and the Office of Science of the Department of Energy. Los Alamos is operated by the Los Alamos National Security, LLC for the US Department of Energy under contract No. DE-AC52-06NA25396.

References

1. Barker, K.J., Davis, K., Hoisie, A., Kerbyson, D.J., Lang, M., Pakin, S., Sancho, J.C.: Entering the Petaflop Era: The Architecture and Performance of Roadrunner. In: SC 2008, Austin, TX (2008)
2. Kahle, J.A., Day, M.N., Hofstee, H.P., Johns, C.R., Maeurer, T.R., Shippy, D.: Introduction to the cell multiprocessor. IBM Journal of Research and Development 49(4), 589–604 (2005)
3. Bowers, K.J., Albright, B.J., Bergen, B.K., Yin, L., Barker, K.J., Kerbyson, D.J.: 0.374 Pflop/s Trillion-particle Particle-in-cell Modeling of Laser Plasma Interactions on Roadrunner. In: ACM Gordon Bell Prize finalist, Supercomputing Conference (SC 2008), Austin, TX (2008)
4. Swaminarayan, S., Kadau, K., Germanm, T.C.: 350-450 tflops molecular dynamics simulations on the roadrunner general-purpose heterogeneous supercomputer. In: ACM Gordon Bell Prize finalist, Supercomputing Conference (SC 2008), Austin, TX (2008)
5. IBM: Data Communication and Synchronization Library for Hybrid-x86: Programmer’s Guide and API Reference. IBM Technical document SC33-8408-00, IBM SDK for Multicore Acceleration version 3, release 0 (2007)
6. Indiana University: Open-MPI (2009), <http://www.open-mpi.org>
7. Barrett, R., Berry, M., Chan, T.F., Demmel, J., Donato, J.M., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., der Vorst, H.V.: Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd edn. SIAM, Philadelphia (1994)
8. Williams, S., Oliker, L., Vuduc, R., Demmel, J., Yelick, K.: Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In: Supercomputing Conference (SC 2007), Reno, NV (2007)
9. Morris, G.R., Prasanna, V.K.: Sparse matrix computations on reconfigurable hardware. IEEE Computer 40(3), 58–64 (2007)
10. Garland, M.: Sparse matrix computations on manycore GPU’s. In: Annual ACM IEEE Design Automation Conference (DAC 2008), Anaheim, CA (2008)
11. Zaki, M.J., Li, W., Parthasarathy, S.: Customized dynamic load balancing for a network of workstations. Parallel and Distributed Computing 43(2), 156–162 (1997)
12. Pineau, J.F., Robert, Y., Vivie, F.: Revisiting matrix product on master-worker platforms. In: Workshop on Advances in Parallel and Distributed Computational Models (APDCM 2007), IEEE International Parallel and Distributed Processing Symposium, Long Beach, CA (2007)
13. Xu, C., Lau, F.: Load Balancing in Parallel Computers: Theory and Practice. Kluwer Academic Publishers, Norwell (1996)
14. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.: StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In: Euro-Par Conference, Delft, The Netherlands (2009)

A Unified Framework for Load Distribution and Fault-Tolerance of Application Servers

Huaigu Wu¹ and Bettina Kemme²

¹ SAP Labs Canada, Montreal, Quebec, Canada

Huaigu.Wu@sap.com

² McGill University, Montreal, Quebec, Canada

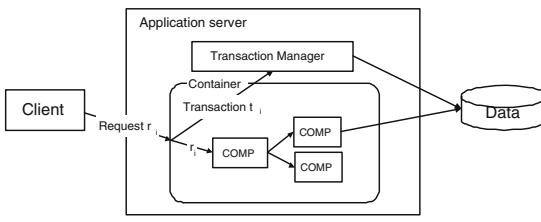
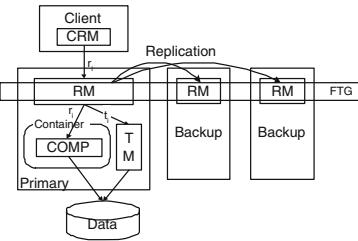
kemme@cs.mcgill.ca

Abstract. This paper proposes a novel replication architecture for stateful application servers that offers an integrated solution for fault-tolerance and load-distribution. Each application server replica is able to execute client requests and at the same time serves as backup for other replicas. We propose an effective load balancing mechanism that is only load-aware if a server is close to become overloaded. Furthermore, we present transparent reconfiguration algorithms that guarantee that each replica has the same number of backups in a dynamic environment where replicas can join or leave at any time. Our evaluation shows that our approach scales and distributes load across all servers even in heterogeneous environments while keeping the overhead for fault-tolerance and load-balancing small.

1 Introduction

Application servers (AS) have become a prevalent building block in current information systems. The AS executes the application programs and maintains volatile data, such as session information, i.e., the server is *stateful*. AS often execute crucial and heavy loaded tasks, and have to handle a large number of concurrent clients, while at the same time have to provide short response times and high reliability. Both requirements can be achieved via replication. Fault-tolerant replication solutions are proposed, e.g., in [1][2][3], while scalability solutions are presented in [4]. However, little research has been performed on providing a combined solution both for scale-out and for high reliability. Similarly, existing AS products such as JBoss and WebLogic offer separate replication solutions for fault-tolerance and load-balancing each having its own setup.

In order to achieve scalability, load balancing algorithms use a cluster of AS replicas, each equipped with the same application software, and distribute request execution across the replicas. Ideally, the more replicas, the higher the maximum throughput the cluster can achieve. In this paper, we refer to a group of replicas executing requests as the *load distribution group* (LDG). In contrast, fault-tolerance algorithms use server replicas to mask the failures of individual replicas. Most commercial solutions let a primary replica execute requests, while the other replicas are backups. The primary sends data changes to backups at critical time points. If the primary fails, a *failover* procedure makes one of the backups the new primary and clients are automatically reconnected to it. For fault-tolerance purposes, it is typically enough to have one or two backups. We refer to a group of one primary and its backups as *fault-tolerance group* (FTG).

**Fig. 1.** Application server architecture**Fig. 2.** Architecture of ADAPT-SIB

A simple approach to combine load balancing and fault tolerance together is to distribute the workload over a set of server replicas, each of them having dedicated backups. However, since backup tasks have typically a low overhead, capacity of the backups is wasted. Furthermore, adding a new replica to handle increased load requires to add backups, and a failed machine needs to be replaced with a new machine to handle future crashes. In order to tackle this problem, we present a unified framework where each replica executes its own client requests and at the same time is backup for other servers. All replicas of an AS cluster belong to a single LDG and can execute client requests. At the same time, each replica is primary of a (small) FTG and is backup in some other FTGs. Our unified solution distributes load across all replicas using a simple and efficient load balancing mechanism that is only load-aware if a server is threaded to become overloaded, handles failures transparently using a dynamic reconfiguration mechanism that automatically guarantees that each replica has a sufficient number of backups at any time, and can easily grow and shrink with the demands.

We use the ADAPT-SIB fault-tolerance algorithm [53] that provides stateful application server replication with strong consistency properties. However, our architecture is also adaptable to other fault-tolerance algorithms. We implemented our solution into the JBoss application server. Our experiments show that the approach efficiently combines fault-tolerance and scalability tasks.

2 Background

2.1 Stateful Application Server

In an AS, the business logic is modularized into different components (e.g., *Enterprise JavaBeans*) as depicted in Figure 1(a). *Volatile components* maintain session information and other related information. They are typically associated with a single client (e.g., Stateful Session Beans). Their state is lost when the AS crashes. In contrast, *persistent components* maintain data that is loaded from the databases (e.g., Entity Beans).

Client requests are calls to interface methods of components. Often, a client has to first connect to the server and after that, a session is created. The volatile components of this client are associated with this session. A client may usually only submit a request once it has received a response for its previous request. Therefore, in general, no concurrency control is needed for volatile components. When persistent components or

the backend database are accessed, requests are normally executed within a transaction that isolates the operations of concurrent transactions.

2.2 Group Communication

We use a *group communication system* (GCS) for communication where the AS replicas are the members of a GCS group. A member can multicast a message to all group members (including itself) or send point-to-point messages. The multicast we use provides *uniform-reliable delivery*. It guarantees that if any member receives a message (even if it fails immediately afterwards) all members receive the message unless they fail. Furthermore, we use multicasts that either deliver messages in FIFO order (messages of the same sender are received in sending order), or in total order (if two members receive messages m and m' , they both receive them in the same order). Furthermore, GCS maintains a view of the currently connected members. Whenever the view configuration changes (through explicit joins or leaves, or due to crashes), the GCS informs all members by delivering a view change message with the new view. Many GCS provide *virtual synchrony* [6]: If members p and q receive both first view V and then V' , they receive the same set of messages while members of V .

2.3 ADAPT-SIB Replication Algorithm

The basic fault-tolerance mechanisms used in our system are based on the ADAPT-SIB algorithm [5][3]. We describe the main steps of ADAPT-SIB necessary to understand the rest of the paper. Figure 2 shows the architecture. ADAPT-SIB assumes there is one primary replica and several backup replicas. All join a single fault-tolerance group FTG (one GCS group). Each AS replica has a *replication manager* (RM) that executes the replication algorithm. At the client, there is a *client replication manager* (CRM).

The CRM has a list of all server replicas. It intercepts each client request and directs it to the current primary replica. If the primary fails, the CRM finds the new primary and resubmits the last request to the new primary if it did not receive a response before the crash. At the primary replica, each request is executed within a transaction. At commit time of a transaction, a checkpoint message containing changes performed on volatile components is multicasted with uniform reliable delivery to all replicas in the FTG. If the primary fails, all active database transactions are aborted by the database system. All remaining backup replicas in the FTG receive a view change message from the GCS and then one of them becomes the new primary. The new primary reconstructs the correct state of volatile components from the checkpoint messages. Then, the new primary receives resubmitted requests and new requests. ADAPT-SIB ensures that the state of the AS and the database are consistent and that the new primary does not execute any request twice. Previously failed or completely new replicas can join in the FTG at runtime. A joining replica receives all checkpoint messages from one existing replica in the FTG and then performs backup tasks. The cost at the backups of receiving the state changes and processing them is only 5-10% of the costs at the primary to execute the original requests. Thus, using ADAPT-SIB for fault-tolerance, each replica has the potential to act as a primary and also as backup for some others.

3 Unified Architecture

In order to exploit the resources of all replicas, we propose an architecture that allows to distribute client requests across all replicas in a cluster and at the same time guarantees fault tolerance using the ADAPT-SIB protocol. All replicas in the cluster are members of a single load distribution group LDG. Each replica is primary in exactly one fault-tolerance group FTG, referred to as its *primary FTG*, and thus, there are as many FTGs as there are replicas in the system. Furthermore, each replica is backup in m FTGs, referred to as the replica's *backup FTGs*. As a result, each FTG has m backups. We refer to this as the m/m *property*. This property allows for a simple, yet powerful automatic reconfiguration mechanism, and also helps in load distribution. In the following, we first discuss how the system starts up, then we explain how load balancing is done, and finally talk about reconfigurations.

3.1 Cluster Initialization

At initialization, one has to decide on the number m of backups in an FTG. Furthermore, one has to indicate the initial number n of replicas in the cluster. The cluster size may shrink or increase later dynamically, but m remains fixed.

Figure 3 shows the final setup after initialization. Each replica uses its address as a unique identifier. At each replica LBM refers to a *Load Balancing Manager* (LBM), *PRM* to the primary replication manager of the primary FTG, and *BRMS* refers to the array of backup replication managers for the backup FTGs. When a replica starts up, its LBM first joins the LDG. Once all n replicas have joined, each LBM multicasts its replica identifier using total-order, uniform-reliable delivery. Each LBM receives the messages in the same order and stores the identifiers in an ordered list, called the replica list *RL* according to the delivery order. Each replica in the system is assigned an *order number* i , $1 \leq i \leq n$, which is the position of the replica in the replica list *RL*. We refer to the replica with order number i as r_i . Note that while the identifier of a replica does not change during its lifetime, the order number might change, as we will see later. Once r_i has determined its order number i , it joins fault-tolerant group FTG_i as primary. If $i > m$, it furthermore joins FTG_{i-m} to FTG_{i-1} as backup. A replica with order $i \leq m$ joins FTG_{n-m+i} to FTG_n and FTG_1 to FTG_{i-1} as backup. For

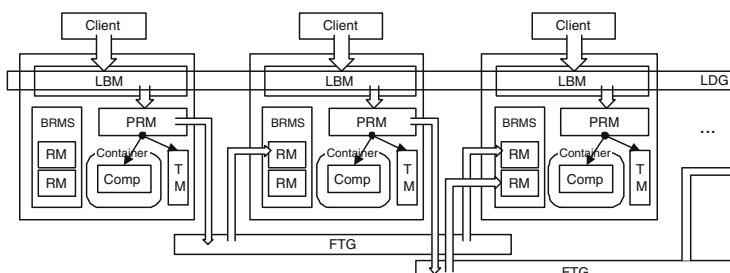
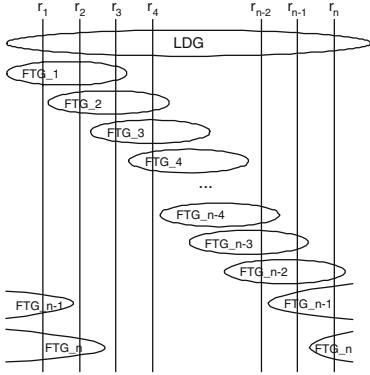
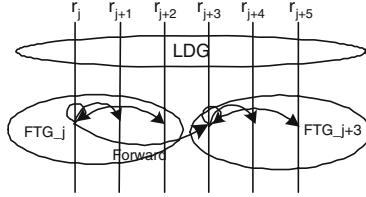


Fig. 3. Unified Architecture

**Fig. 4.** Cluster Initialization**Fig. 5.** Forwarding a request

instance if $m = 2$, then r_3 joins FTG_1 and FTG_2 , r_2 joins FTG_n and FTG_1 , and r_1 joins FTG_{n-1} and FTG_n . Figure 4 depicts this circular setup of FTGs.

3.2 Load Balancing

Simple Load Distribution. When a client connects to the system, a session on one primary replica is created, and all requests within this session are handled by this replica. This guarantees that each request sees the current session state. Thus, load balancing is performed at a connection time. It goes through two phases. The client has a pre-defined replica list CL with potential server replicas (received, e.g., when looking up the service on a registry). It submits connection requests to these replicas until it receives a response. When an available replica r_i receives such a request it becomes the load-balancer for this request. The LBM of r_i randomly decides on a replica r_j from its replica list RL to serve the client, and returns the client replication manager CRM object to the client. The message piggybacks a failover list containing the identifiers of all members of FTG_j (derived from replica list RL) and indicates that r_j is the primary. The failover list FL is used by the client CRM for fault tolerance, and is described in more detail in Section 3.3. In the second phase, the newly installed CRM sends a session request to r_j . Then r_j accepts client requests within the session.

Our load-balancing algorithm is very efficient, as the message overhead for the session setup is the same as in standard AS architectures involving one message round for the connection request (with r_i), and one for session creation (with r_j). Scheduling decisions are made locally at the server site without the need to exchange and maintain load information. The replica list CL does not need to be accurate or complete as the sessions themselves are equally distributed among all active replicas.

Load Forwarding. Random replica assignment, however, does not work well in heterogeneous environments or if request execution times are not uniform. For example, if a server gets a few heavy requests it will be temporarily overloaded and any further request will deteriorate the performance. We address this issue with a simple but effective

first-local-then-forward (FLTF) mechanism which leads to request forwarding if the load of a replica is above a given threshold. Load could be measured as memory usage, CPU usage, response time, or the number of connected clients. We refer to a replica with a load below the threshold as a *valid* replica.

When the LBM of replica r_j receives a client session request and its load is below the threshold, it serves the client directly as described above. Otherwise, r_j multicasts a load query message lqm to all replicas in FTG_j in order to find a valid replica. (see left ellipse in Figure 5). Upon receiving lqm a backup returns a positive answer if its load is below the threshold. r_j chooses the first r_k to answer as the one to serve the client. It returns to the CRM the list of replicas belonging to FTG_k . The CRM refreshes its local failover list FL , and sends a session request to r_k . In case of isolated overloads, contacting m other nodes will likely find a node that can accept new clients. However, if there is no positive answer among the backups, r_j sends a *forward* message to the replica with the smallest order number larger than any order number in FTG_j , i.e., $r_{(j+m+1)\%n}, r_{(j+m+1)\%n}$ now repeats the process, sending a new load query message lqm in its own primary $FTG_{(j+m+1)\%n}$. Figure 5 shows how a forward is sent to the primary of FTG_{j+3} if $m = 2$. If a valid replica is found, $r_{(j+m+1)\%n}$ returns the information to r_i so that it can forward the relevant replica list to the client. Otherwise, an additional forward could take place. If after a maximum number of T iterations no valid replica is found, a negative message is sent to the originator r_j which either accepts or refuses the client. If T is set 0, then no forward message is sent at all. Setting T low makes sense because if all nodes in r_j 's neighborhood are saturated, it is likely that the entire system is close to saturation and further forwarding will not help.

Discussion. A main benefit of our load distribution algorithm is that it is purely distributed without any central controller and can be easily implemented. Load messages are only sent if a node is overloaded, load is checked in real time, and replicas can individually decide to take on further load or not. It does not affect the fault-tolerance algorithm but takes advantage of the FTG infrastructure.

One question is how often one has to forward to find a lightly loaded replica. The probability of finding a valid replica within the m backup replicas of the local FTG is equal to the probability of finding a valid replica within any m replicas in the cluster. If there are k valid replicas randomly distributed in the cluster, the probability p of finding one of the k valid replicas within the m backup replicas is $p = 1 - (n-m-1)/k$. Since each forward searches $m+1$ replicas (a new FTG), the probability p of finding one of k valid replicas within the m backups of the initiator and T further forwards is $p = 1 - (n-m-1-T*(m+1))/k$. Assume the cluster has 100 replicas and $m = 2$. With 50 valid replicas and $T = 1$ we find a valid replica with more than 97% probability. With only 20 valid replicas and $T = 3$, the probability is 92.8%. Thus, this simple mechanism provides a high success rate even for highly loaded clusters with low message overhead.

3.3 Reconfiguration

In this section, we show how the system automatically maintains the m/m property (each FTG has m backups, each replica is backup of m FTGs) which guarantees that each replica has at any time enough backups.

Server crash. When a replica r_i fails it leaves FTG_i , and a new primary has to be found for r_i 's clients. Furthermore r_i is removed as a backup from m FTGs. Thus, these FTGs need new backups. For simplicity of notation the following discussion assumes that a replica r_i fails where $i > m$ and $i + m < n$.

The failover process at the server side is slightly different from the original ADAPT-SIB protocol. No new primary can be built for FTG_i , since the backups in FTG_i have their own primary FTGs. Instead, the clients associated with FTG_i will be migrated to FTG_{i+1} , and FTG_i is removed. The main reconfiguration steps are as follows.

- r_{i+1} , which is a backup in FTG_i , becomes the new primary for r_i 's clients. It first performs the failover for the sessions of clients in FTG_i as described in ADAPT-SIB. Then, it makes these clients part of FTG_{i+1} . Finally, it leaves FTG_i .
- The other backups in FTG_i (r_{i+2} to r_{i+m}) must also migrate the session information of the old clients of FTG_i to FTG_{i+1} . After the migration, they also leave FTG_i . As a result, FTG_i does not have any member anymore.
- r_{i+m+1} was not member of FTG_i but now needs to receive the session information of the clients that were migrated from FTG_i to FTG_{i+1} . The primary of FTG_{i+1} sends this information to r_{i+m+1} .
- Since replicas r_{i+1} to r_{i+m} have left FTG_i , they are now backups for only $m-1$ FTGs. Furthermore, FTG_{i-m} to FTG_{i-1} only have $m-1$ backups since r_i was removed from these groups. To resolve this, r_{i+1} joins FTG_{i-m} as backup, r_{i+2} joins FTG_{i-m+1} , etc, as the joining process of new replicas described in ADAPT-SIB.
- Finally, each load balancing manager LBM removes r_i from its replica list RL , and decreases the order numbers of replicas r_{i+1} to r_n by one.

Server recovery. When a new or failed replica joins in a cluster of size n , it first joins the single load distribution group LDG, and all replicas are notified about this event. Each replica adds the new replica with order number $n + 1$ to its replica list RL and considers it in its load-balancing task. r_{n+1} receives the replica list RL from a peer replica. According to our setting, r_{n+1} must have a primary FTG and m backup FTGs.

- r_{n+1} creates a new FTG_{n+1} and joins it.
- r_{n+1} joins FTG_{n-m+1} to FTG_n as backups. These FTGs have now $m + 1$ backups.
- Now, r_1 to r_m leave FTG_{n-m+1} to FTG_n respectively. The FTGs are now back to having m backups.
- Finally, r_1 to r_m join the new FTG_{n+1} . They have again m backup FTGs and FTG_{n+1} has m backups. The recovery is fast, since this group is new.

The reconfiguration is complete and r_{n+1} starts accepting client requests.

4 Experiments and Evaluation

We integrated our approach into the JBoss application server (v. 3.2.3) [7]. For this paper, we use a micro benchmark where each client request performs operations on stateful session beans associated with the client but the database is not accessed. This allows us to isolate the replication effects on the AS. Clients connect to the system

and then run for 10 seconds continuously submitting requests. All requests trigger transactions with similar load. Further experiments can be found in [8]. Unless otherwise stated, experiments were performed on a cluster of 64-bit Xeon machines (3.0 GHz and 2G RAM) running RedHat Linux. As GCS we use Spread [9]. In all our settings, each FTG consists of one primary and two backups.

4.1 Experiment 1: Basic Performance

We have a first look at the performance of our unified architecture when no replicas leave or join the system. In the figures, JBoss refers to a standard single-node non-replicated JBoss application server without fault-tolerance. ADAPT-SIB refers to a system running the ADAPT-SIB algorithm, i.e., there is one fault-tolerance group FTG but no load distribution group LDG. ADAPT-LB refers to the approach proposed in this paper with one LDG using our load-balancing approach and several FTGs running ADAPT-SIB. JC/RoundRobin refers to a replicated cluster using JBoss' own round-robin based load-balancer. This configuration does not provide any fault-tolerance.

Figure 6 shows response times with increasing number of clients injected in the system per second for three machines. In the non-replicated JBoss and ADAPT-SIB clients compete soon for resources and response times deteriorate quickly. ADAPT-SIB has higher response times than a non-replicated JBoss, since the primary has to perform the replication. In contrast, ADAPT-LB and JC/RoundRobin have low response times up to 15 clients due to load distribution. Each node is less loaded and can provide faster service. While ADAPT-LB has higher response times than JC/RoundRobin the difference is smaller than between ADAPT-SIB and the non-replicated JBoss, because ADAPT-LB is able to distribute the fault-tolerance overhead across all replicas.

Scalability is further analyzed in Figure 7 which shows that the throughput for ADAPT-LB and JC/RoundRobin increases linearly with the number of replicas. Due to fault-tolerance activity on each node, ADAPT-LB achieves less throughput than JC/RoundRobin. But even JC/RoundRobin does not provide ideal throughput since load-balancing has its own overhead.

In summary, ADAPT-LB truly provides both fault-tolerance and scalability.

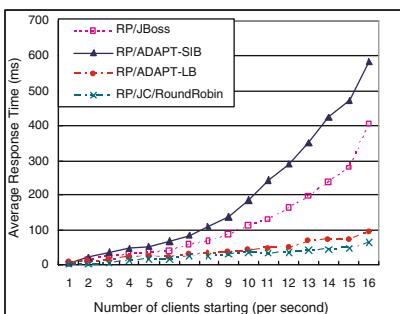


Fig. 6. Response Times with 3 Replicas

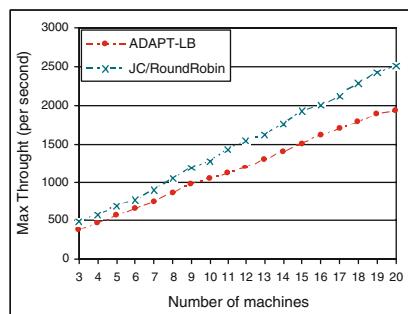


Fig. 7. Scale-up

4.2 Experiment 2: Heterogeneity

Heterogeneity is a challenge for load-balancing techniques. We first analyze the impact of heterogeneous hardware by replacing half of the machines with PIII machines (850 MHz and 256M RAM). Figure 8 shows the maximum achievable throughput with increasing number of machines for ADAPT-LB with forwarding (ADAPT-LB/FLTF), ADAPT-LB using only random load-balancing without forwarding (ADAPT-LB/Random), and JBoss' round-robin load-balancer (JC/RoundRobin). In general, the throughput is lower than in the homogeneous environment (Fig. 7), since half of the machines are now weaker. ADAPT-LB/Random is the worst because random assignment ignores heterogeneity and fault-tolerance adds overhead. ADAPT-LB/FLTF and JC/RoundRobin have similar performance despite the fact that ADAPT-LB has the fault-tolerance overhead. A detailed analysis has shown that, compared to JC/RoundRobin, ADAPT-LB/FLTF has lower throughput on the weak and higher throughput on the strong nodes. This is because with ADAPT-LB/FLTF weak nodes forward requests that are then executed by the strong nodes. Thus, ADAPT-LB/FLTF compensates the overhead of fault-tolerance by a smarter load-balancing strategy which assigns more tasks to the stronger nodes.

Another type of heterogeneity are dynamic workload changes. In the next test, we use an additional very heavy client transaction with an average response time of 2000 ms to simulate the heterogeneous workload. We only compare ADAPT-LB/FLTF and JC/RoundRobin. At the beginning of this test, a cluster consisting of 6 identical machines runs the micro benchmark for 30 seconds. Then we artificially inject the heavy transaction into the system. We refer to the machine executing the heavy transaction as *HC*. The other are denoted as *LC*. Figure 9 has as x-axis time slots of 100 ms. The heavy transaction starts at timeslot 5. Before injecting the heavy transaction, *HC* and *LC* have the same response times which are higher for ADAPT-LB because of the fault-tolerance overhead. Using JC/Round-Robin, the *HC* response times increase to around 400 ms after the injection because the *HC* machine becomes saturated. The response times on the *LC* group remain the same because they are not affected. Using ADAPT-LB, response times on the *HC* machine increase for the first 5 time slots after the heavy transaction is injected. This represents transactions of clients that were already

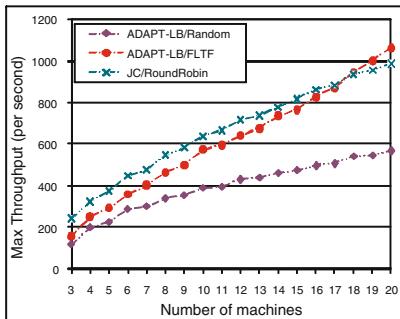


Fig. 8. Heterogeneous hardware

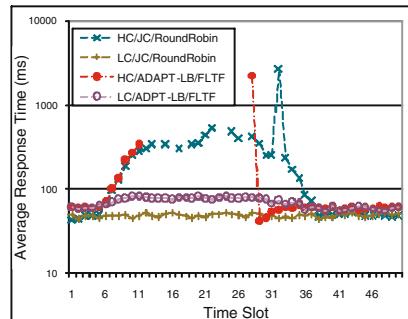


Fig. 9. Heterogeneous workload

assigned to HC when the long transaction arrived. Then there is a long gap, since HC does not accept any further clients anymore according to the forwarding strategy. At timeslot 27 the long transaction finishes (with a long response time). After that HC accepts clients providing standard response times for them. While the heavy transaction is running on HC we observe longer response times at the LC machines because HC redirects clients to them, and thus they are more loaded. In total, response times are less affected with ADAPT-LB than with JC/RoundRobin which has unacceptable high response times for some of the clients.

Clearly, our approach can quickly respond to the temporary overload of individual machines. The extra overhead only has to be paid when overload occurs.

4.3 Experiment 3: Failover and Recovery

We now show the behavior of the system during and after reconfiguration. A cluster of 6 replicas (r_1, \dots, r_6) runs the micro benchmark for about 30 seconds when replica r_3 crashes. We distinguish three types of replicas. NP (new primary) indicates the replica r_4 that takes over the clients of the failed replica r_3 . B indicates replicas that have to reconfigure their backups (r_5 and r_6). NI indicates all other replicas on which the failure has no impact (r_1 and r_2). Figure 10(a) has as x-axis time slots of 100 ms, and as y-axis the average response time within a time slot. The crash occurs at time slot 5.

Before the crash, the average response time is similar in each group. After the crash, the response time on the new primary NP drastically increases because it requires considerable resources to perform the failover. This process takes about 300-400 ms. After that, the average response time is still higher than on the other groups because NP has now double the clients. The response time in the replicas that have to reconfigure their backups (B) also increases (it shortly doubles) because the state transfer to the new backups takes some of the resources. The recovery process to become a backup takes less than 100 ms. However, the response time on B remains higher and actually also increases on NI for which no reconfiguration is necessary. The reason is that there is now one less replica in the system to execute requests. Furthermore, since NP is still higher loaded, the replicas in B and NI accept more of the newly injected clients.

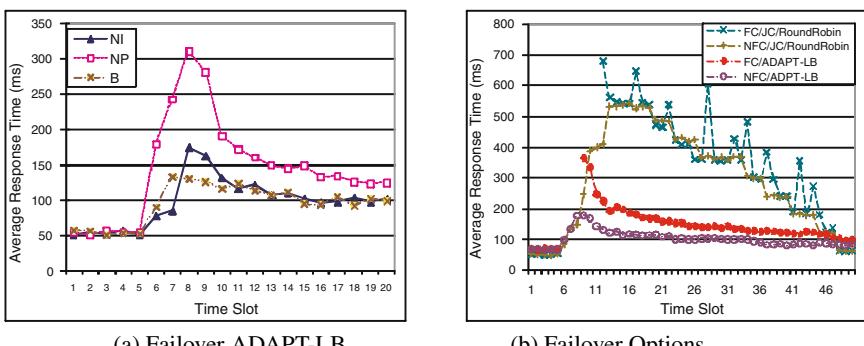


Fig. 10. Reconfiguration

Eventually, once NP has stabilized, the system becomes balanced again. The average response time on all remaining replicas converges eventually to the same value. This value is now higher because there is one less replica in the system to serve requests.

We further conducted a test where we added a 7th replica to a running system. The join shortly doubled the response times of those clients that were connected to replicas who had to change their $FTGs$. But the system reconfigured in less than 400 ms. After that response times are generally lower as load was now distributed over more replicas.

As a final experiment, we compare ADAPT-LB with an alternative solution. Using JBoss' round-robin architecture, when a replica crashes, each client originally connected to the failed replica connects to any of the correct replicas and resubmits all requests from the beginning of the session. We call this solution the *re-execution solution*. Note that this only works correctly if the requests do not trigger changes on permanent components because these changes are already in the database and should not be applied again. We again use 6 machines and crash one replica at time slot 5. This time, we group response times by client type. FC clients were originally connected to the crashed replica, and NFC are all other clients. Figure 10(b) shows the average response time over time. In ADAPT-LB, one replica takes all FC clients (and has additionally NFC clients). As long as this new primary performs failover, the FC are blocked. Therefore, there is a gap for FC clients where no response times are measured, and once execution resumes there is a peak in average response times. NFC clients are also affected, but much less (as discussed before). Response times for both FC and NFC quickly go back to normal levels. As long as the new primary is more loaded, the forwarding strategy distributes new clients to other replicas.

In the re-execution solution, re-executing the historical requests is a heavy task that takes place at all replicas. For FC , the replay takes at least 10 time slots where no response is created. But response times stay high for *all* clients for a long time and only go down gradually because the machines are overloaded with the replay process. A peak in the graph of the FC clients occurs when one of them finishes the failover process as this is the time response times are measured. This shows that if replicas should be used for both load distribution and fault-tolerance then it is paramount to have a fast failover procedure as provided by ADAPT-SIB in order to keep the system responsive during failover times. A replay solution seems too expensive.

In summary, our approach can handle failures and recovery transparently and dynamically. Reconfiguration affects the client response times only shortly, and is relatively localized to few machines.

5 Related Work

Load balancing and fault-tolerance have traditionally be handled as orthogonal issues, and research on one topic usually does not attempt to solve the other. For fault tolerance of application servers, most industrial (e.g., JBoss, Weblogic or WebSphere) and many research solutions (e.g., [2][3]) use the primary-backup approach.

Typical load balancing solutions of application or web servers use a central load balancer. As we have seen, content-blind policies [10], such as Random or Round Robin do not work well in heterogeneous environments. Content-aware policies (e.g., [4][11][12])

require some knowledge about the environment, such as load or access patterns which can have considerable overhead. In contrast, our approach is purely distributed, does not maintain state information, has little overhead, and is easy to implement.

Only a few approaches consider both load distribution and fault-tolerance. Singh et al. [13] propose a system that merges the Eternal fault tolerance architecture [1] and the TAO load balancer [4]. A similar architecture is used in [14]. All servers in a cluster are partitioned to several disjoint FTG groups. Only the primary server in each replica group is used for load balancing. Moreover, these solutions do not address reconfiguration problems. [15] organize replicas in a chain and execute updates on the head (who then propagates the changes down the chain) while reads are executed on the tail. However, client requests are distributed over head and tail which is problematic for stateful application servers. Also, not all replicas are evenly utilized.

Scalability and fault-tolerance is also an issue in file-systems. For instance, Coda [16] also distributes load over servers and uses replication for availability. But files are always persistent and there is no backend database. Also, Coda does not follow the primary-backup replication but clients take care to update all copies.

6 Conclusion

This paper describes a new replication architecture for stateful application servers that offers both fault-tolerance and load balancing in a single integrated solution. Replication is completely transparent to the clients, all resources in the system are used, and the system requires little intervention by administrators. The solution is simple to implement yet powerful. The architecture is completely distributed. Our implementation shows that our approach increases the scalability even in heterogeneous environments, and provides dynamic reconfiguration in a dynamic environment with little overhead.

References

1. Narasimhan, P., Moser, L.E., Melliar-Smith, P.M.: Strongly consistent replication and recovery of fault-tolerant CORBA applications. *Journal of Computer System Science and Engineering* 32(8) (2002)
2. Barga, R., Lomet, D., Weikum, G.: Recovery guarantees for general multi-tier applications. In: *Int. Conf. on Data Engineering, ICDE* (2002)
3. Wu, H., Kemme, B.: Fault-tolerance for stateful application servers in the presence of advanced transactions patterns. In: *IEEE Symp. on Reliable Distrib. Systems, SRDS* (2005)
4. Schmidt, D.C., Levine, D.L., Mungee, S.: The design of the TAO real-time object request broker. *Computer Communications* 21(4) (1998)
5. Wu, H., Kemme, B., Maverick, V.: Eager replication for stateful J2EE servers. In: *Int. Symp. on Distributed Objects and Applications, DOA* (2004)
6. Chockler, G.V., Keidar, I., Vitenberg, R.: Group communication specifications: A comprehensive study. *ACM Computing Surveys* 33(4) (2001)
7. Fleury, M., Reverbel, F.: The JBoss extensible server. In: *Middleware* (2003)
8. Wu, H., Kemme, B.: A unified framework for load distribution and fault-tolerance of application servers. Technical Report SOCS-TR-2009.1, McGill University (2009)
9. Amir, Y., Danilov, C., Miskin-Amir, M., Schultz, J., Stanton, J.: The Spread toolkit: Architecture and performance. Technical Report CNDS-2004-1, Johns Hopkins Univ. (2004)

10. Andreolini, M., Colajanni, M., Morselli, R.: Performance study of dispatching algorithms in multi-tier web architectures. *SIGMETRICS Performance Evaluation Review* (2002)
11. Pai, V.S., Aron, M., Banga, G., Svendsen, M., Druschel, P., Zwaenepoel, W., Nahum, E.: Locality-aware request distribution in cluster-based network servers. In: *Int. Conf. on Architectural Support for Programming Languages and Operating Systems* (1998)
12. Liu, X., Zhu, X., Padala, P., Wang, Z., Singhal, S.: Optimal multivariate control for differentiated services on a shared hosting platform. In: *IEEE Conf. on Decision and Contr.* (2005)
13. Singh, A.V., Moser, L.E., Melliar-Smith, P.M.: Integrating fault tolerance and load balancing in distributed systems based on CORBA. In: Dal Cin, M., Kaâniche, M., Pataricza, A. (eds.) *EDCC 2005. LNCS*, vol. 3463, pp. 154–166. Springer, Heidelberg (2005)
14. Othman, O., Schmidt, D.C.: Optimizing distributed system performance via adaptive middleware load balancing. In: *ACM SIGPLAN Workshop on Optimization of Middleware and Distributed Systems* (2001)
15. Renesse, R.V., Schneider, F.B.: Chain replication for supporting high throughput and availability. In: *Symp. on Operating System Design and Implementation, OSDI* (2004)
16. Satyanarayanan, M., Kistler, J.J., Kumar, P., Okasaki, M.E., Siegel, E.H., Steere, D.C.: Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. Computers* 39(4), 447–459 (1990)

On the Feasibility of Dynamically Scheduling DAG Applications on Shared Heterogeneous Systems^{*}

Aline P. Nascimento¹, Alexandre Sena¹, Cristina Boeres²,
and Vinod E.F. Rebello²

¹ Institutos Superiores de Ensino La Salle, Niterói, RJ, Brazil

² Instituto de Computação, Universidade Federal Fluminense, Niterói, RJ, Brazil
`{deapaula,asena,boeres,vinod}@ic.uff.br`

Abstract. Grid and Internet Computing have proved their worth executing large-scale *bag-of-task* class applications. Numerous middlewares have been developed to manage their execution in either dedicated environments or opportunistic and shared ad-hoc grids. While job dependencies are now being resolved by middleware capable of scheduling workflows, these environments have yet to be shown beneficial for message passing parallel applications. Obtaining high performance in these widely available environments without rewriting existing parallel applications is of up most importance to *e-Science*. The key to an efficient solution may be an alternative execution model and the efficient dynamic scheduling of application processes. This paper presents a hierarchical scheme for dynamically scheduling parallel DAG applications across a set of non-dedicated heterogeneous resources. In order to efficiently tackle process dependencies and adapt to varying system characteristics, dynamic schedulers are distributed *within* the application and operate in a *collaborative* and *pro-active* fashion to keep overheads low.

1 Introduction

The increasing interest in *e-Science* is driving the development of an ever growing number of tightly coupled (message passing) applications that demand large scale execution environments. With programmers focusing on the large scale parallelisation of their problems, they continue to assume a dedicated homogeneous execution environment. Given that gaining access to sufficient compute time on individual HPC clusters can be difficult, or that acquiring and maintaining such systems do not come cheap, many scientists are being forced to turn to alternative less specialised aggregated environments. While the quantity of resources may now no longer be an issue, these environments are not considered conducive to existing HPC applications since the resources and communication links may be heterogeneous and shared with other applications.

^{*} This work is supported by research grants from CNPq and FAPERJ.

Although many parallel applications are potential candidates to execute in such distributed large scale environments, extracting high performance from this type of platform is not trivial, especially for non-experts. One promising approach is the design of autonomic applications, applications with the ability to manage their own execution without user influence, i.e. become *self-configuring*, *self-healing*, *self-optimising* and *self-protecting* [12]. This paper focuses on the implementation of the dynamic scheduling system that supports the *self-optimising* property of *EasyGrid Application Management System* [11] enhanced MPI applications. Even in dynamically changing environments, *EasyGrid* applications become aware of the system conditions, detect suboptimal behaviour and make optimisation decisions to adapt their execution.

Being hierarchical, the scheduling scheme allows different policies to be applied at each level. However, unlike other previous work in this context, the proposed dynamic scheduling strategy uses a novel *collaborative* and *pro-active* approach. The schedulers at each level solve different scheduling problems. Thus collaboration is required between schedulers distributed in adjacent levels of the hierarchy to achieve a single objective: the best possible execution time for the application given the computational power available; or complete the execution within a predetermined time frame, for example. No central scheduler exists to make unilateral scheduling decisions, schedulers cooperate and negotiate the transfer of processes for rescheduling. Unlike traditional dynamic task schedulers that react by only allocating ready processes to idle resources, the more efficient pro-active strategy [10] reschedules both ready and pending (still awaiting data) processes in advance of resources becoming idle. The latter scheme effectively permits processes to be *rescheduled* across individual resource task queues while the former generally only schedules processes, from a central queue, once. These combined characteristics aims to offer the *EasyGrid AMS* better scalability in large distributed environments, while keeping scheduling overheads low.

To the best of our knowledge, no such dynamic scheduling scheme has been implemented specifically for parallel applications with task dependencies in grid middleware. An efficient scheduling policy specifically for autonomic *bag-of-tasks* applications was presented in [11] and a comparison between intra-site scheduling heuristics for tightly coupled application in [10]. This paper looks at the impact of a hybrid scheduling approach [4] that has been adopted to integrate the benefits of both static and dynamic heuristics. A static scheduler can afford to analyse the application as a whole, in depth, this cost does not adversely impact the execution time of the application, although precise performance information about the target architecture is required. The hybrid approach tends to improve the quality and to reduce the cost of scheduling decisions [14], since in addition to up to date system information, the dynamic scheduler now has useful information about the application obtained from the static schedule.

2 The EasyGrid AMS and Its Execution Model

The *EasyGrid AMS* is implemented as a wrapper to the LAM/MPI library, thus an existing MPI application need only be recompiled in order to be transformed

into an autonomic version – no modification to the original code is required. Nor does any additional middleware need to be installed on grid resources. The execution of MPI application’s processes are controlled by a three level hierarchical management system that is composed of the following entities: a single *Global Manager* (GM), at the top level, which manages the inter-site activities of the application in the distributed system; at each site, a *Site Manager* (SM) is responsible for the allocation of processes to the resources available at the site; and finally, the *Host Manager* (HM), one for each resource, takes on the responsibility for the creation and execution of the MPI processes on that host.

In this work, the *application model* is based on the class of parallel applications that can be represented by directed acyclic graphs (DAGs). A task graph is denoted by $G = (V, E, \varepsilon, \omega)$, where: the set of vertices V represents *tasks*; E , the precedence relation among them; $\varepsilon(v)$ is the amount of work associated with task $v \in V$; and $\omega(u, v)$ is the weight associated with the edge $(u, v) \in E$, representing the amount of data units transmitted from task u to v . The topological level of a task v , denoted as *level*(v), is defined as $\max_{u \in \text{pred}(v)} \{\text{level}(u) + 1\}$, given that the level of any source task is zero and $\text{pred}(v)$ is the set of immediate predecessors of v . The *architectural model* specifies the main features of the target architecture. Given the set R of available heterogeneous computational resources (processors), cp_j is the *computational power* of each resource $r_j \in R$ and the *communication delay index*, $cdi_{i,j}$, estimates the latency cost associated with each communication link (r_i, r_j) . Both are constantly updated during the application’s execution by the *EasyGrid AMS*.

The *EasyGrid AMS* follows a non-traditional MPI execution model [15] that executes one MPI process per task instead of executing one long running MPI process per processor as in the traditional approach. In this approach, finer-grained processes with precedence relationships between them are formed. Although this increases the number of processes, if managed carefully, the additional costs associated with dynamic process creation, message re-routing and message logging, can be offset by the benefits of not requiring computationally expensive process checkpointing and migration techniques [15].

3 Dynamic Scheduling

In order to create a flexible and scalable strategy, the proposed dynamic scheduling approach has a hierarchical structure as seen in Figure 1. Associated with each management process, the dynamic schedulers at each level perform distinct functions and collaborate with others in adjacent levels of the hierarchy. The idea is to divide and distribute the decision making mechanisms to reduce both the complexity and the overheads of the scheduling problem and to react faster to changes by applying corrective measures locally, at the first instance. Furthermore, each dynamic scheduler can have its own policy and different application-specific heuristics to better adapt the application to the system as a whole and the availability of individual resources. Note that the MPI processes are created dynamically during the execution of the application (and not all at once at start

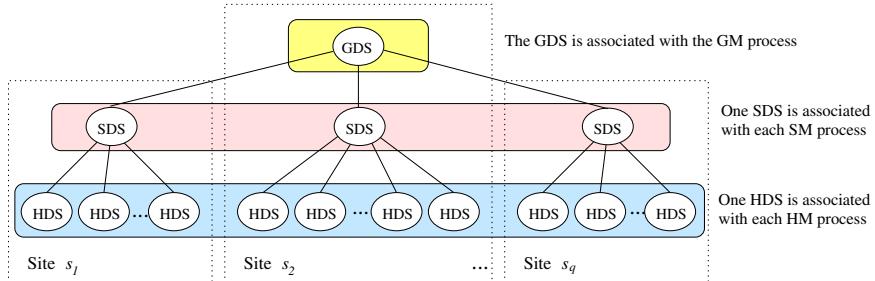


Fig. 1. The hierarchical scheduler architecture of the *EasyGrid AMS*

up) by the dynamic schedulers in the HMs. Only tasks which have not been created (i.e. are not in execution) may be rescheduled.

The dynamic schedulers are divided into the following entities: *Global Dynamic Scheduler* (GDS), *Site Dynamic Scheduler* (SDS) and *Host Dynamic Scheduler* (HDS). The GDS is associated with the GM process and it is responsible for rescheduling tasks between different sites. Each site manager has its own SDS, which is responsible for rescheduling tasks among site resources. Finally, at the bottom level, each HM runs a HDS_j that creates user processes in accordance with the application's topology and the distributed system local access policies. Collectively, the dynamic schedulers estimate the remaining execution time of the application and consequently, the makespan of the actual schedule.

In order to minimise the overhead associated with the scheduling decisions, dynamic schedulers only analyse a subset or block of tasks in V at each *scheduling event* [4]. A block of tasks B_l contains the tasks $v \in V$ with $level(v) \leq l$ that have not been created yet, where l is given by the highest topological level of a task already in execution, plus one. This allows task v to be considered for rescheduling more than once. With regard to the size of B_l , while a large block will increase the intrusion overhead due to number of tasks that must be evaluated, a small block provides a very limited preview of the remaining tasks.

3.1 The Host Dynamic Scheduler - HDS

When starting the execution of the application, each resource $r_j \in R$ receives the list of tasks and their execution order, as specified by a static scheduler. Each HDS_j is responsible for three local scheduling policies: the *ordering*, the *concurrency* and the *resource access policies*. If activated, and respecting the precedence constraints, the ordering policy may or may not permit HDS_j to change the execution order of tasks in response to environmental variations. For example, suppose that v_i and v_l are two independent tasks and that v_i is statically scheduled before v_l in resource r_j . If during the application execution, at a given time v_l becomes ready before v_i due to a faster execution not predicted by the static scheduler, re-ordering the execution of these tasks may allow the HDS_j to improve performance by not delaying v_l . While scheduling algorithms

traditionally only consider the execution of one task at a time on each processor, it is often more efficient to execute more than one concurrently. The concurrency policy specifies the number of application tasks that may execute simultaneously on r_j . The number depends on the characteristics of both the application tasks (e.g., granularity, I/O requirements) and the resource (e.g., existing workload, number of processor cores) [11]. The resource access policy is determined by the resource owner and defines when r_j may be used by this application (e.g., only when the system load is less than 20%, or only between 6pm and 8am).

Upon completion of a task in r_j , the wall clock and CPU execution times are collected by the AMS monitoring layer and made available to HDS_j . With these values in hand and the *computational slowdown index* csi_j , a value which is inversely proportional to a normalised metric of the benchmarked processing capacity of r_j , the HDS_j estimates its effective computational power cp_j and the estimated remaining time ert_j to execute the current set of the ready tasks allocated to r_j . The HDS_j also verifies the time when r_j will be available to execute the next ready task, $ready_j$. These three values are added to the monitoring message and sent to the respective SDS_k to be used by the site dynamic scheduler. The values sent to the SDS_k should be reasonably up to date, so if the time elapsed since the last monitoring message exceeds a certain limit, the values can also be estimated from system calls. The HDS_j also collaborates with its respective SDS_k when receiving a request for tasks for reallocation or when receiving new tasks which have been allocated to r_j .

3.2 The Site Dynamic Scheduler - SDS

Let $S = \{s_1, s_2, \dots, s_q\}$ be the set of q sites in a target grid and R_k be the set of resources in site s_k , where $R_k \subseteq R$. The SDS_k associated with s_k is responsible for rescheduling tasks among its resources in order to minimise the *site execution time*. Particular to this work, each SDS_k treats ready and pending tasks *distinctly* in order to apply different priorities and scheduling event frequencies. In the case of ready tasks, since these tasks are independent only a simple heuristic is required [11]: when receiving the ert_j and cp_j values for resources in s_k , the SDS_k calculates the target *site estimated execution time* ($sert_k^*$) for the remaining ready tasks in s_k and the *site imbalance index* (sii_k), which represents the degree of load imbalance across the resources. A scheduling event will be triggered if sii_k rises above a predefined threshold. The details of the task selection and reallocation processes were presented in [10].

In the case of the pending tasks, the heuristic first identifies the resource $r_{max} \in R_k$ that determines the *site execution time*. The SDS_k employs the following mechanisms while there exists a resource r_{max} which can have its finish time minimised: (1) identifies and solicits critical predecessor tasks from other resources or (2) transfers tasks to an under loaded resources in s_k . In the first mechanism, it is possible for the SDS_k to request a critical task that belongs to another site. In this case, the GDS will be act as a mediator. Note however that a request for a task from another site s_l , may be denied if rescheduling that task adversely affects the site execution time of s_l .

3.3 The Global Dynamic Scheduler - GDS

The GDS also behaves distinctly in relation to *ready* and *pending* tasks. In case of *ready* tasks the scheduling mechanism performed by the GDS is similar to the one used by each SDS_k , allowing the GDS to trigger global scheduling events whenever it deems necessary. Whenever a predefined minimum time interval expires, the GDS calculates if the task allocation needs to be adjusted. Considering the *average estimated remaining time* and the sum of the *computational powers* associated with each site s_k , the GDS calculates the target *global estimated remaining time (gert*)* to execute the ready tasks and the *system imbalance index (sii)*. A scheduling event will be triggered, if sii_k is above a predefined threshold.

In a scheduling event, the GDS determines the set of sites that should receive (S_{sub}) and also those that should release tasks (S_{ask}). The GDS sends the *gert** value to each $s_k \in S_{ask}$ such that the associated SDS_k selects the tasks to be re-scheduled based on this value. All ready tasks with estimated finish time greater than *gert** will be requested by their respective SDS. After receiving the requested tasks from each SDS_{ask} , the GDS distributes them among the sites $s_k \in S_{sub}$, which in turn will allocate them to their local resources accordingly.

As the GDS does not maintain the computational power of the grid resources but only an estimated average of the site, scheduling events are not triggered for pending tasks by the GDS. Instead the GDS acts as a mediator between sites for task transfer requests, comparing the *site execution times* and verifying if the re-scheduling should be performed. This approach was implemented because the cost of keeping up to date status information for all the available resources would be not scalable for large scale distributed environments.

4 Performance Analysis

The benefits of the *pro-active* and *collaborative* mechanisms implemented in the dynamic hierarchical scheduler of the hybrid approach are analysed when executing DAG parallel applications in heterogeneous and shared platforms. An experimental evaluation was performed in a dedicated environment with controlled background workloads to emulate the dynamism of computations on a shared grid. A set of 24 Pentium IV 2.6 GHz processors with 512Mb RAM, running Linux Fedora Core 2, Globus Toolkit 2.4 and LAM/MPI 7.0.6 was used.

Four classes of DAGs, at different granularities, were chosen in order to consider the impact of degrees of parallelism and tasks dependencies. These include synthetic parallel MPI applications in the form of binary *out-trees*, binary *in-trees* and *diamond* graphs, denoted by OUT_n , IN_n and DI_n , respectively, with n being the number of tasks. These synthetic applications allow the amount of work executed, and data communicated, by each MPI process to be parameterised. A real astrophysics application that simulates the evolution of a system composed of N bodies or particles, with Newton's gravitational forces being exerted on each body due to the interaction with other bodies in the system, was also considered. The results presented are the arithmetic average of at least 8 execution times, which turned out to always be reasonably close.

4.1 Relative Efficiency of the Hybrid Approach

For a performance analysis of the dynamic scheduling architecture, synthetic MPI applications managed by the *EasyGrid AMS* were executed under four distinct scheduling scenarios, given that the distributed system available was composed of two sites with 11 resources each, with one site offering only 50% of its computational power, and the other 100% for the duration of the experiments. The four scenarios were: (1) static schedules generated by the well known HEFT heuristic [16] based on precise information about the available computational power of the resources in this static heterogeneous environment. The dynamic scheduler was deactivated; (2) Identical to scenario (1), but with the dynamic scheduler activated; (3) Without performance information, HEFT assumes that the resources have the same computational power. However, the dynamic scheduler makes adjustments at runtime; (4) Identical to scenario (3), but with the dynamic scheduler deactivated. In the four scenarios, the synthetic application were composed of tasks with uniform computational costs (two instances of each application with task durations of 1 and 5 seconds on the fastest resource, respectively), and message sizes of 128 bytes.

Table 1 presents the makespans for each scenario and a comparison between them. The last three columns indicate the net benefits, for a completely static environment, of hybrid scheduling over static scheduling, with and without performance information. The execution times obtained by the static approach in scenario (1) establish an *ideal value* for the applications given: no overheads, associated with the dynamic schedulers, are incurred; and the good static allocation provided by HEFT using precise performance information.

Considering the tree-like applications in scenarios (1) and (2), the results suggest that, despite short term scheduling decisions and overheads, the proposed dynamic scheduling approach manages to obtain a performance very close to the *ideal values*. Note that while one should expect scenario (1) to present better results, in practice, variations occur in the execution time even for the same code running on identical resources. Thus even in a static environment, the dynamic scheduler will try to take advantage of the opportunity to improve the execution time. In the case of the diamond DAG, which has a higher degree of dependencies between tasks and a smaller degree of parallelism, the makespans are slightly higher (for tasks of 5s, a degradation of 9%). As the dynamic scheduler

Table 1. Makespans in a static heterogeneous environment

t_{exe}	DAG	(1)	(2)	(3)	(4)	% (2)/(1)	% (4)/(1)	% (3)/(2)	% (3)/(4)
1s	OUT_{4095}	266.81	265.33	275.39	397.59	0.55%	-49.02%	-3.79%	30.74%
	IN_{4095}	269.23	267.92	281.86	406.44	0.49%	-50.96%	-5.20%	30.65%
	DI_{4096}	315.88	330.29	411.91	481.18	-4.56%	-52.33%	-24.71%	14.40%
5s	OUT_{4095}	1281.99	1287.22	1310.21	1931.87	-0.41%	-50.69%	-1.79%	32.18%
	IN_{4095}	1287.44	1292.61	1343.79	1940.34	-0.40%	-50.71%	-3.96%	30.74%
	DI_{4096}	1430.41	1558.75	1832.96	2315.60	-8.97%	-61.88%	-17.59%	20.84%

re-schedules a limited size block B_l of tasks at each scheduling event, decisions are taken without considering the effects on successor tasks in subsequent blocks.

In practice, obtaining precise performance information prior to execution is not easy. The third last and second last columns (without and with dynamic scheduling respectively) emphatically highlight the degradation in performance when precise information is not available. However, the AMS and its dynamic scheduler's ability to make adjustments bring significant benefits. The lack of knowledge about the resources causes the faster resources to become under utilised and diminishes overall performance as seen by the results for scenarios (3) and (4). Without dynamic scheduling, the degradation is between 49% and 62%. The higher degree of parallelism in the tree applications offers more opportunities for the dynamic schedulers to make the appropriate corrections in the task allocations. With a higher degree of dependencies among tasks, it is harder for the dynamic schedulers to completely take advantage of the idle times when scheduling the diamond DAG. Nonetheless, improvements of up to 21% and more than 30% for the tree DAGs were achieved.

4.2 Heterogeneous Shared Environments

To evaluate the performance of the dynamic scheduler in shared heterogeneous environments, the following experiment evaluates the same three scheduling scenarios (1), (2) and (3) defined in Subsection 4.1, this time with extra CPU-bound processes being launched in a controlled manner to emulate a shared system. Based on the execution times of scenario (1) in Table 1, denoted as et , an additional load was inserted at time $1/4 \times et$ on the resources of the slower site so that only 33% of its computational power was available. Then, at $3/4 \times et$, all extra loads on this site were extinguished so that 100% of computational power was made available. For scenarios (1) and (2), the static scheduler has precise performance information about the initial state of the resources, but is unaware if and when the available computational power will change. As presented in Table 2, the best results were obtained for the scenarios (2) and then (3), in which the dynamic scheduler was active. Given the difficulties for the static scheduler to predict load changes in the system, the hierarchical dynamic scheduler manages to re-schedule tasks appropriately, improving performance. While

Table 2. Makespans in a shared heterogeneous environment

t_{exe}	DAG	(1)	(2)	(3)	$\%(2)/(1)$	$\%(3)/(2)$
1s	OUT_{4095}	357.53	268.22	270.02	33.30%	-0.67%
	IN_{4095}	357.19	281.33	291.46	26.97%	-3.60%
	DI_{4096}	415.52	373.42	375.72	11.27%	-0.62%
5s	OUT_{4095}	1708.44	1262.94	1280.50	35.27%	-1.39%
	IN_{4095}	1710.37	1300.34	1300.46	31.53%	-0.01%
	DI_{4096}	1960.64	1546.09	1581.41	26.81%	-2.28%

the second last column shows improvements between 11% and 35%, obtained by the dynamic scheduler, the last column indicates that although precise performance information at compile time is valuable, its importance for the overall execution is diminished even for diamond DAGs.

4.3 A Real Application: An N -Body Simulation

Based on the traditional MPI execution model with one process v_k per processor $r_j \in R$ for the duration of the application, the classic ring algorithm for the N -body application [9] executes as follow: each process calculates the forces between $N/|R|$ particles at each one of the $|R|$ stages (loop iterations). After each stage, the process sends $N/|R|$ particles to one of its neighbours and, before initiating the next stage, receives $N/|R|$ particles from the other. Note that the efficiency of ring algorithm comes from the fact that all processes have the same number of particles and computation progresses in a synchronous pipelined fashion and thus well suited to execution in homogeneous stable environments.

A grid enabled ring algorithm was proposed in [14] based on the alternative execution model [15] that executes one process per task. In addition to the message to its neighbour, each process sends a message to its successor process. This ring application is modelled as a mesh-like DAG with a width and height equal to the degree of parallelisation, W . The optimal performance of the algorithm depends on the ideal process granularity (related to W), an appropriately chosen subset of the available resources, and good schedule for the W^2 tasks. A detailed analysis and an innovative strategy to find the ideal W and the appropriate subset of heterogeneous resources using HEFT can be found in [14].

This next experiment evaluates the performance of the two versions when coping with a dynamic shared environment. The traditional approach is denoted as MPI ring while the new one, AMS ring, employs the dynamic scheduler. A short series of time steps were executed here, although solutions for real astrophysical problems typically require thousands of time steps. The 24 resources were divided in three sites of 8 resources each, with an extra workload executed in one resource, chosen randomly, in the first site, offering the ring application only 50% of that resource's computational power. After 1/3 of the application's total execution time, this workload migrates to a resource in the second site, again chosen randomly. The same behaviour occurs after 2/3 of the execution time when this extra workload migrates to a resource site 3.

The execution times of the two ring algorithms are presented in Figure 2 together with a theoretical lower bound (shown in the graph) for increasing numbers of time steps (TS). This lower bound was calculated based on the total available computational power; assuming perfect load balancing; and that communication costs were zero. Note that in each of five experiments, there are only two external workload changes. As expected, the MPI ring algorithm's poor relative performance is caused by the inability to modify its process allocation or its own degree of parallelism, even though the number of processes was efficiently derived in accordance with the strategy proposed in [14].

The improvement over the MPI ring can be seen in the last column of the table. In accordance with [14], the best number of processes derived for a 200,000 particle instance was $W^2 = 48^2$, which indeed achieved the best performance. Under the alternative execution model, the AMS ring approach exercises its application management system to near optimal effect - the actual execution times are within 4.5% of a lower bound which ignores all operating system and AMS overheads. The fact that a few competing jobs can lead to almost 80% worse than ideal performance is one of the reasons why this class of application was previously not considered apt for non-dedicated grids.

5 Related Work

Most of the work on developing management systems for grid environments, such as Nimrod/G [1], Condor/G [7] and MyGrid [6], has focused on *bag-of-tasks* applications, characterised by independent tasks and commonly executed on grids. With regard to the scheduling heuristics, most systems adopt dynamic scheduling policies to cope with the heterogeneity and dynamic behaviour of grid resources. Some well known heuristics are: *workqueue*, *workqueue with replication*, *sufferage*, *Max-Min*, and *Min-Min* [3]. Additionally, there are grid management systems that adopt scheduling heuristics based on a grid economy [2]. In this case, the scheduling policies consider the total amount that a user is willing to pay to have its application executed in a determined interval of time.

Some grid management systems, e.g. GrADS [5] and GridWay [8], are able to execute parallel applications with task dependencies. Research into mapping DAG applications to heterogeneous systems has received intense attention, e.g. [16]. However, most dynamic scheduling approaches address the problem by iteratively load balancing ready tasks. Adaptive self-scheduling schemes adjust the iterative load allocations according to changing processing capabilities [13].

In relation to the scheduling structure, it is usual to find grid management systems that adopt a centralised approach, where a central scheduler is responsible for scheduling the tasks of all applications in the distributed system or all the applications of a single user. Some known examples are GrADS [5], MyGrid [6] and Nimrod/G [1]. On the other hand, there are management systems

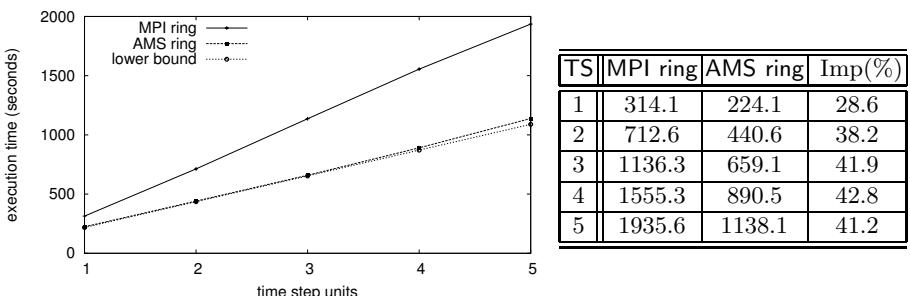


Fig. 2. Performance of MPI ring and AMS ring in a shared heterogeneous environment

that utilise a decentralised scheduling approach, where each application may have its own scheduler and adopt specific policies such as GridWay [8].

6 Conclusions

This work presents a novel dynamic scheduling strategy that treats the precedence relationships that may exist in parallel applications. The scheduler is part of an application management system (AMS) which is embedded in the user's MPI application thus bestowing self-scheduling properties. The *EasyGrid AMS* employs a distributed array of cooperating schedulers, each with different functions depending on their position in the hierarchy, but which collaborate to achieve a single objective - minimising the makespan. The schedulers at each level also need not be homogeneous; each is individually capable of employing different application specific policies that may even change during execution.

For entirely static (i.e. typical HPC) environments, if precise performance data is available, a static good scheduler will be able to extract excellent performance. However, as the imprecision of this information increases, so will the degradation in the quality of the schedule. The feasibility of an effective dynamic scheduling strategy depends on low intrusion. Here, distributed dynamic schedulers and hybrid scheduling (with the dynamic schedulers adjusting a pre-defined static schedule), have been employed to address these issues. In comparison to a static scheduling only approach, results show that the hybrid approach has acceptable performance in spite of the overheads and unnecessary scheduling changes. However, the benefits are significantly clearer when the available performance data is not accurate. In shared environments, a dynamic scheduling approach is fundamental. In the case of the *EasyGrid AMS*, precise performance data can be useful but is not essential to achieve good performance. The approach has been applied to scientific applications with a grid enabled (i.e. *EasyGrid AMS* enhanced) version of the tightly-coupled parallel N -body ring application executing efficiently in a heterogeneous non-dedicated computational grid.

References

1. Buyya, R., Abramson, D., Giddy, J.: Nimrod/G: An architecture for a resource management and scheduling system in a global computational grid. In: Proc. 4th International Conf. on High Performance Computing in the Asia-Pacific Region, vol. 1, pp. 283–289. IEEE Computer Society Press, Los Alamitos (2000)
2. Buyya, R., Abramson, D., Venugopal, S.: The grid economy. Proceedings of the IEEE, Special Issue on Grid Computing 93(3), 698–714 (2005)
3. Casanova, H., Legrand, A., Zagorodnov, D., Berman, F.: Heuristics for scheduling parameter sweep applications in grid environments. In: Proc. 9th Heterogeneous Computing Workshop (HCW 2000), pp. 349–363. IEEE Computer Society Press, Los Alamitos (2000)
4. Choudhury, P., Kumar, R., Chakrabarti, P.P.: Hybrid scheduling of dynamic task graphs with selective duplication for multiprocessors under memory and time constraints. IEEE Trans. Parallel Distributed Systems 19(7), 967–980 (2008)

5. Berman, F., et al.: New grid scheduling and rescheduling methods in the GrADS project. *International Journal of Parallel Programming* 33(2), 209–229 (2005)
6. Cirne, W., et al.: Running bag-of-tasks applications on computational grids: The MyGrid approach. In: Proc. 32nd International Conference on Parallel Processing (ICPP 2003), pp. 407–416. IEEE Computer Society Press, Los Alamitos (2003)
7. Frey, J., Tannenbaum, T., Livny, M., Foster, I., Tuecke, S.: Condor-G: A computational management agent for multi-institutional grids. *Journal of Cluster Computing* 3(5), 237–246 (2002)
8. Huedo, E., Montero, R.S., Lorente, I.M.: The GridWay framework for adaptive scheduling and execution on grids. *Scalable Computing: Practice and Experience* 6(3), 1–8 (2005)
9. Makino, J.: An efficient parallel algorithm for $O(N^2)$ direct summation method and its variations on distributed-memory parallel machines. *New Astronomy* 7(7), 373–384 (2002)
10. Nascimento, A.P., Boeres, C., Rebello, V.E.F.: Dynamic self-scheduling for parallel applications with task dependencies. In: Proc. 6th International Workshop on Middleware for Grid Computing. ACM Press, New York (2008)
11. Nascimento, A.P., Sena, A.C., Boeres, C., Rebello, V.E.F.: Distributed and dynamic self-scheduling of parallel MPI grid applications. *Concurrency and Computation: Practice and Experience* 19(14), 1955–1974 (2007)
12. Parashar, M., Hariri, S.: Autonomic Computing: Concepts, Infrastructure, and Applications. CRC Press, Boca Raton (2007)
13. Riakotakis, I., Ciorba, F.M., Andronikos, T., Papakonstantinou, G.: Self-adapting scheduling for tasks with dependencies in stochastic environments. In: Proc. 5th International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (HeteroPar 2006), pp. 1–8. IEEE Computer Society Press, Los Alamitos (2006)
14. Sena, A.C., Nascimento, A.P., Boeres, C., Rebello, V.E.F.: EasyGrid enabling of iterative tightly-coupled parallel MPI applications. In: Proc. IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA 2008). IEEE Computer Society Press, Los Alamitos (2008)
15. Sena, A.C., Nascimento, A.P., da Silva, J.A., Vianna, D.Q.C., Boeres, C., Rebello, V.E.F.: On the advantages of an alternative MPI execution model for grids. In: Proc. 7th IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2007), pp. 575–582. IEEE Computer Society Press, Los Alamitos (2007)
16. Topcuoglu, H., Hariri, S., Wu, M.Y.: Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems* 13(3), 260–274 (2002)

Steady-State for Batches of Identical Task Trees

Sékou Diakité¹, Loris Marchal², Jean-Marc Nicod¹, and Laurent Philippe¹

¹ Laboratoire d’Informatique de Franche-Comté
Université de France Comté, France

² Laboratoire de l’Informatique du Parallelisme
CNRS - INRIA - Université de Lyon, France

Abstract. In this paper, we focus on the problem of scheduling batches of identical task graphs on a heterogeneous platform, when the task graph consists in a tree. We rely on steady-state scheduling, and aim at reaching the optimal throughput of the system. Contrarily to previous studies, we concentrate upon the scheduling of batches of limited size. We try to reduce the processing time of each instance, thus making steady-state scheduling applicable to smaller batches. The problem is proven NP-complete, and a mixed integer program is presented to solve it. Then, different solutions, using steady-state scheduling or not, are evaluated through comprehensive simulations.

1 Introduction

Computing Grids gather large-scale distributed and heterogeneous resources, and make them available to large communities of users [1]. Such platforms enable large applications from various scientific fields to be deployed on large numbers of resources. These applications come from domains such as high-energy physics [2], bioinformatics [3], medical image processing [4], etc. Distributing an application on such a platform is a complex duty. As far as performance is concerned, we have to take into account the computing requirements of each task, the communication volume of each data transfer, as well as the platform heterogeneity: the processing resources are intrinsically heterogeneous, and run different systems and middlewares; the communication links are heterogeneous as well, due to their various bandwidths and congestion status.

Applications are usually described by a (directed) graph of tasks. The nodes of this graph represent the computing tasks, while the edges between nodes stand for the dependencies between these tasks, which are usually materialized by files: a task produces a file which is necessary for the processing of some other task. In this paper we consider *Grid jobs* made of a collection of input data sets that must all be processed by the same application. We thus have several instances of the same task graph to schedule. Such a situation arises when the same computation must be performed on independent data [5] or independent parameter sets [6]. Moreover, the targeted applications we plan to schedule do not include any replication phases in the process. Hence, DAGs considered in this paper have no fork nodes, and consists in chains or in-trees. This corresponds to application like medical [7] or media [8] image processing workflows.

The problem consists in finding a schedule for these task trees which minimizes the overall processing time, or makespan. This problem is known to be NP-hard. To overcome this issue, some of us proposed to use steady-state scheduling [9]. In steady-state

scheduling, we assume that instances to be performed are so numerous that after some initialization phase, the flow of computation will become steady in the platform. By characterizing resource activities in this *steady state*, we are able to derive a periodic schedule that maximizes the *throughput* of the system, that is the number of task graph instances completed within one time unit. As for makespan minimization, this schedule is asymptotically optimal. This means that for a very large number of instances to process, the initialization and clean-up phases that wrap the steady-state phase become negligible, and the makespan of the steady-state schedule becomes close to the optimal. However, when the number of instances is important but bounded, existing steady-state approaches do not give optimal performances – initialization and clean-up phases cannot be neglected when scheduling a finite number of instances – and lead to a huge number of ongoing instances. In this paper, we propose an adaptation of the steady-state scheduling that allows to use it on batches of jobs of finite size, without compromising its asymptotically optimality.

The rest of the paper is organized as follows. In Section 2 we give a short reminder on the steady-state techniques and their drawbacks. In Section 3 we formalize the problem we are dealing with, and assess its complexity. In Section 4 we propose an exact solution to this problem. Simulations showing its impact are reported in Section 5.

2 Steady-State Scheduling for Task Graphs

2.1 Platform and Application Model

In this section, we detail the model used in the following study. First, we denote by $G_P = (V_P, E_P)$ the undirected graph representing the platform, where $V_P = \{P_1, \dots, P_p\}$ is the set of all processors. The edges of E_P represent the communication links between these processors. The time needed to send a unit-size message between processors P_i and P_j is denoted by $c_{i,j}$. We use a bidirectional one-port model: if processor P_i starts sending a message of size S to processor P_j at time t , then P_i cannot send any other message, and P_j cannot receive any other message, until time $t + S \times c_{i,j}$.

The application is represented by a directed acyclic graph (DAG) $G_A = (V_A, E_A)$, where $V_A = \{T_1, \dots, T_n\}$ is the set of tasks, and E_A represents the dependencies between these tasks, that is, $F_{k,l} = (T_k, T_l) \in E_A$ is the file produced by task T_k and consumed by task T_l . The dependency file $F_{k,l}$ has size $\text{data}_{k,l}$. We use an unrelated computation model: computation task T_k needs a time $w_{i,k}$ to be entirely processed by processor P_i .

We assume that we have a large number of similar task graphs to compute. Each instance is described by the same task graph G_A , but has a different input data from the others. This corresponds to the case when the same computation has to be performed on different input data sets.

2.2 Principle

In this section, we briefly recall steady-state techniques and their use for task graph scheduling. The steady-state approach has been pioneered by Bertsimas and

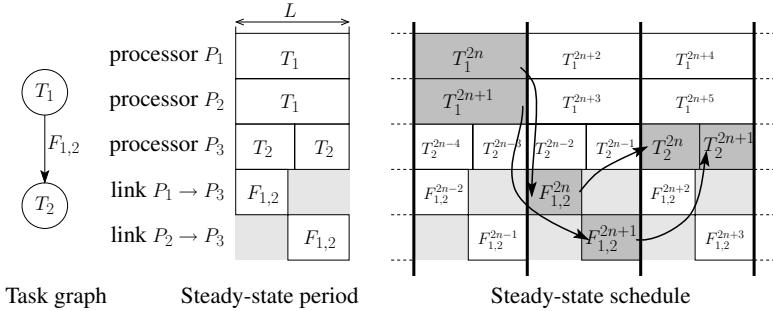


Fig. 1. Handling dependencies

Gamarnik [10]. The present study is based on a steady-state approach for scheduling collections of identical task graphs proposed in [9]. The steady state is characterized using activities variables: α_i^k represent the average number of tasks T_k processed by processor P_i within one time unit in steady state. We similarly define activities for data transfers: $\beta_{i,j}^{k,l}$ represent the average number of files $F_{k,l}$ sent by P_i to P_j within one time unit in steady state.

By focusing in the steady state, we can write constraints on these activity variables, due to speed limitation of the processors and links. We also write “conservation laws” to state that files $F_{k,l}$ have to be produced by tasks T_k and are necessary to the processing of tasks T_l . We obtain a set of constraints that totally describe a valid steady-state schedule. We add the objective of maximizing the throughput, that is the overall number of DAGs processed by time unit, to get a linear program. Solving this linear program over the rational numbers allows us to compute the optimal steady-state throughput.

Then, from an optimal solution of this linear program, we construct a periodic schedule that achieves this optimal throughput. The construction of this schedule is complex, especially for handling communications, and we refer the interested reader to [9] for a detailed description. In the solution of linear program, the average number of tasks (or files) processed (or transferred) in a time unit may be rational. However, we cannot split the processing of a task, or the transfer of a file, into several pieces. Thus, we compute the lowest common multiple L of all denominators of these quantities. We then multiply all quantities by L , to get a period where every quantity of tasks or files is integer. A period describes the activity of each processor (how many task of each types is performed) and of each link: communications are assembled into groups that can be scheduled simultaneously without violating the one-port model constraints. In the following, we will consider these communications groups as one special task, assigned to a fictitious processor P_{p+1} ; a dependency between a task T and a file F is naturally transformed into a dependency between T and the special task representing the group of communication which contains the file transfer F .

Although bounded, the length L of the period may be large. The steady-state schedule is made of a pipelined succession of periods, as described in Figure 1. Dependencies between files are taken into account when reconstructing the schedule: a file $F_{k,l}$ produced by T_k during period 1 will be transferred to another processor during period 2

and then used by task T_l during period 3. Figure 11 describes a steady-state schedule obtained for a simple task graph: in a period both processors P_1 and P_2 process a task T_1 , while P_3 processes two tasks T_2 , achieving a throughput of 2 instances every L time units. In the periodic schedule, each task or file transfer is provided with its instance number in superscript, and dependencies are materialized with arrows for instances $2n$ and $2n + 1$.

Once the periodic schedule is built, it can be used to process any number of tasks. A final schedule consists of three phases:

1. an initialization phases, where all the preliminary results needed to treat a period are pre-computed;
2. the steady-state phase, composed of several periods;
3. a clean-up phase, where all remaining tasks are processed so that all instances are completed.

2.3 Shortcomings

We have seen that the length of the period of the steady-state schedule may be quite large, and that a large number of periods may be needed to process a single task graph in steady state. This induces a number of drawbacks:

Long latency. For a given task graph, the time between the processing of the first task and the last task, also called latency, may be large since several periods are necessary to process the whole instance. This may be a drawback for interactive applications.

Large buffers. Since the processing time of each instance is large, a large number of instances must be started before the first one is completely processed. Thus, at every time step, a large number of ongoing jobs have to be stored in the system, and the platform must provide large buffers to handle all temporary data.

Long initialization and clean-up phases. Since the length of the period is large and contains many task graph instances, the number of tasks that must be processed before entering steady state is large. Thus, the initialization phase will be long. Similarly, after the steady-state phase, many tasks remain to be processed to complete the schedule, leading to a long clean-up phase. As these phases are done using some heuristic scheduling algorithms, their execution time might be far from the optimal, leading to poor performance of the overall schedule.

In spite of these drawbacks, we have shown in [11] that steady-state scheduling is of practical interest as soon as the number of task graph instances is large enough. In this study, we aim at reducing this threshold, that is to obtain a steady-state schedule which is also interesting for small batches of task graphs.

One could envision two solutions to overcome these drawbacks: (i) decrease the length of the period, or (ii) decrease the number of periods necessary to process one instance. However, there is limited hope that the first solution could be implemented if we want to reach the optimal throughput: the length of the period directly follows from the solution of the linear program. In this study, we focus on the second one, that is on reducing the latency of the processing of every instances.

3 Problem Formulation and Complexity

3.1 Motivation

We aim at scheduling identical DAGs (in-trees) on an heterogeneous platform with unrelated machines. Our typical workload consists of a few hundreds of DAGs. When the period of the steady-state is small compared to the length of the steady-state phase, initialization and clean-up phases are short, and steady-state scheduling is a very good option. When the period obtained is large compared to the steady-state phase, it is questionable to use steady-state as initialization and clean-up may render the advantage of steady-state unprofitable. The overall metric is the time needed to process all the DAGs (total *makespan*). By using steady-state scheduling, we focus on *throughput* maximization. It is possible to get a solution with optimal throughput [9]; our goal is to refine this solution to make it profitable for small batches of DAGs.

The solution proposed in [9] consists of a periodic schedule. The *length of the period* of this schedule is a key parameter for our objective. However, since we want to keep an optimal throughput, we do not try to reduce this length, but we prohibit any increase in the period length, which would go against our final objective.

We have seen that a large number of periods may be needed to completely process one instance. More precisely, after building the steady-state period, each dependency in the task graph can be satisfied within a period, or between two consecutive period, as illustrated in Figure 2. In this figure, we have taken the period of Figure 1, and we have modified its utilization of the schedule, so that one dependency can be satisfied within a period: in the new schedule, the results of file transfer $F_{1,2}$ can be used by task T_2 immediately, in the same period, instead of waiting for the next period. This is done by reorganizing the period: the “first” transfer $F_{1,2}$ of a period is now used to compute the “second” task T_2 . We say that $F_{1,2} \rightarrow T_2$ is an *intra-period* dependency, contrarily to other dependencies that are *inter-period*. Of course, this single modification has little impact on the total makespan, but if we could transform all inter-period dependencies into intra-period dependencies (or a large number), our objective would be greatly improved.

The *number of inter-period dependencies*, that is the dependencies which originate in one period and terminate in the following one, is an important factor. The number

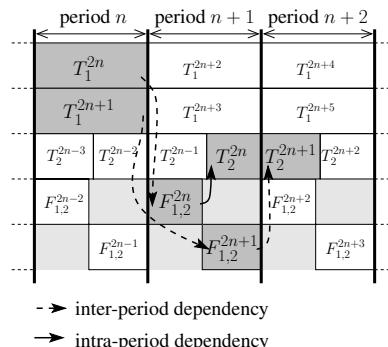


Fig. 2. A periodic schedule with inter-period and intra-period dependencies

of periods needed to completely process an instance (and thus the latency) strongly depends on the number of such dependencies. As for the makespan, the number of instances that have to be started in the initialization phase, and finished in the clean-up phases is exactly the number of inter-period dependencies. Thus, reducing the number of such dependencies is an important goal in order to overcome the drawbacks of the original steady-state implementation. Note that in the original version of the steady-state schedule, the number of these dependencies is huge: all dependencies are inter-period dependencies.

In order to get a practical implementation of steady-state scheduling for bounded sets of task graphs, we choose to forget about direct makespan minimization. We start from a period with optimal throughput (computed as in [9]), and focus on reducing the number of inter-period dependencies in the schedule.

3.2 Formalization of the Problem

We start from the description of a steady-state period. A period consists in q instances of the task graph G_A . The u^{th} instance of task T_k is denoted T_k^u . We call $\sigma(P_i)$ the set of instances of tasks processed by processor P_i . For sake of simplicity, we denote by w_k^u the duration of T_k^u , that is $w_k^u = w_{i,k}$, with $T_k^u \in \sigma(P_i)$.

Dependencies between task instances naturally follows the edges of the task graph: for each edge $T_k \rightarrow T_l \in E_A$, for all $u = 1, \dots, q$, we have a dependency $T_k^u \rightarrow T_l^u$.

The period is provided with a length L , which must not be smaller than the occupation time of any processor: $\sum_{T_k^u \in \sigma(P_i)} w_k^u \leq L$ for all P_i .

The solution to our problem consists in starting times $t(T_k^u)$ for each instance of task T_k^u . We must ensure that two tasks scheduled on the same processor do not overlap:

$$\begin{aligned} \forall P_i, \forall T_k^u, T_l^v \in \sigma(P_i), \text{ with } t(T_k^u) \neq t(T_l^v), \\ t(T_k^u) \leq t(T_l^v) \Rightarrow t(T_k^u) + w_k^u \leq t(T_l^v) \end{aligned} \quad (1)$$

The number of inter-period dependencies for a given solution can be easily computed. A dependency $T_k^u \rightarrow T_l^u$ is an intra-period dependency if and only if T_k^u finishes before the beginning of T_l^u , that is if

$$t(T_k^u) + w_k^u \leq t(T_l^u) \text{ with } T_l^u \in \sigma(P_i). \quad (2)$$

Thus, inter-period dependencies are all dependencies that do not satisfy this criterion.

3.3 Complexity of the Problem

In this section, we assess the complexity of the problem presented in the previous section, namely the ordering of the tasks on each processor, with the objective of minimizing the number of inter-period dependencies.

We first define the decision problem associated to the minimization of the number of inter-period dependencies.

Definition 1 (INTER-PERIOD-DEP). *Given a period described by σ , consisting in q instances of a task graph G_A (which is a tree), on p processors, with computation times*

given by w , and an integer bound B , is it possible to find starting times t (T_k^u) for each task instance such that the resultant number of inter-period dependencies is not larger than B ?

This problem is NP-complete. The proof of this result, based on a reduction from the 3-PARTITION problem, is available in the companion research report [12].

4 Optimal Algorithm with MIP Formulation

In this section, we present a linear program to solve the problem presented in Section 3. This linear program makes use of both integer and rational variables, hence it is a Mixed Integer Program. Solving a MIP is NP-complete, however efficient solvers exist for this problem [13], which makes it possible to solve small instances.

In the following, we assume for the sake of readability that we have only one instance of the task graph in the period. Furthermore, we denote by w_j the processing time of T_j on the processor which executes it. Our approach can be extended to an arbitrary number of instances, at the cost of using more indices.

For any pair of tasks (T_j, T_k) executed on the same processor (that is such that $T_j, T_k \in \sigma(P_i)$ for some P_i), we define a binary variable $y_{j,k}$. We will ensure that $y_{j,k} = 1$ if and only if T_j is processed before T_k .

We also add one binary variable $e_{j,k}$ for each dependency $T_j \rightarrow T_k$. This binary variable expresses if the dependency is an intra-period dependency ($e_{j,k} = 1$) or an inter-period dependency ($e_{j,k} = 0$).

Finally, we use the starting time t_j of each task T_j as a variable. We now write constraints so that these variables describe a valid period.

- We ensure that the y variables correctly define the ordering of the t_j ’s:

$$\forall P_i, \forall T_j, T_k \in \sigma(P_i), \quad t_j - t_k \geq -y_{j,k} \times L \quad (3)$$

$$y_{j,k} + y_{k,j} = 1 \quad (4)$$

- We also check that a given dependency is an intra-period dependency if and only if $e_{j,k} = 1$:

$$\forall T_j \rightarrow T_k, \quad t_k - (t_j + w_j) \geq (e_{j,k} - 1) \times L \quad (5)$$

- We make sure that no task is processed during the processing of task T_j , that is during $[t_j, t_j + w_j]$:

$$\forall P_i, \forall T_j, T_k \in \sigma(P_i), t_k - (t_j + w_j) \geq (y_{j,k} - 1) \times L \quad (6)$$

- Finally, we check that all tasks are processed within the period:

$$\forall T_j, \quad t_j + w_j \leq L \quad (7)$$

Together with the objective of minimizing the number of inter-period dependencies (i.e., maximizing the number of intra-period dependencies), we get the following MIP:

$$\left\{ \begin{array}{l} \text{Maximize } \sum e_{j,k} \\ \text{under the constraints (3), (4), (5), (6) and (7)} \end{array} \right. \quad (8)$$

We can prove that the previous linear program computes a valid schedule with a minimal number of inter-period dependencies (see the companion research report for details [12]).

5 Experimental Results

In this Section, we present experimental results that show how minimizing the inter-period dependencies improves the original steady-state algorithm. We compare four algorithms that schedule batches of identical jobs on a heterogeneous platform. The first algorithm is the original steady-state implementation and the second algorithm the steady-state implementation with the optimization using mixed integer programming to minimize the number of inter-period dependencies described above (called steady-state+MIP). The third algorithm is also a steady-state implementation with inter-period dependencies minimization, but we replace the MIP optimization with a simple greedy algorithm; we do not detail this algorithm, called steady-state+heuristic, but refer the interested reader to the extended research report [12]. The fourth algorithm is a classical list-scheduling algorithm based on HEFT [14]: as soon as a task or a communication is freed of its dependencies, the algorithm schedules it on the resource that guarantees the Earliest Finish Time (EFT). The EFT evaluation depends on the load of the platform and takes both the computation time and the communication time into account. Note that in steady-state strategies, the initialization and clean-up phases are implemented using this list-scheduling technique.

5.1 Simulation Settings

In the following, we report simulation results obtained with a simulator implemented above SimGrid and its MSG API [15]. The experiences consist in the simulation of 245 platform/application scenarios for batches from 1 to 1000 jobs. The platforms are randomly generated; parameters allows platforms from 4 to 10 nodes. Each nodes can process a subset of the 10 different task types with a computing cost between 1 and 11 time units. Network links generation ensures that the platform graph is connected, links bandwidth are homogeneous. The applications are also randomly generated, generation parameters allows in-trees with 5 to 15 nodes. Nodes types are selected between the 10 different task types. Dependency generation ensures that the application graph is an in-tree, dependency file sizes vary from 1 to 2.

For some of the scenarios, large periods and large numbers of dependencies may arise and the optimal dependency reduction with the MIP becomes too costly to compute, even though an efficient MIP solver is used (CPLEX [13]). In the following, we thus distinguish two cases: SIMPLE scenarios are the ones when we are able to solve the MIP (139 scenarios), and GENERAL scenarios gathers all cases, and we do not include MIP results (245 scenarios).

5.2 Number of Inter-Period Dependencies

In the simulations, we count the number of inter-period dependencies that the different strategies (MIP or heuristic) are able to transform into intra-period dependencies.

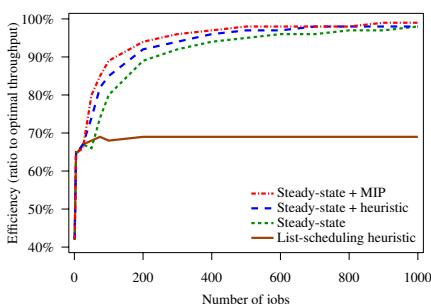
When we are able to solve the MIP, it suppresses 39% of the inter-period dependencies, whereas the heuristic is able to suppress only 29% to 30% of them (29% in all cases, and 30% in SIMPLE cases). This shows that both the MIP and the heuristic strategies achieve a good performance for our metric. As we have outlined in the introduction, this does not necessarily result into an improvement for the global behavior of the schedule. Thus, we also compare the performance of these strategies on other metrics, namely the obtained throughput and the number of running instances.

5.3 Scheduling Efficiency

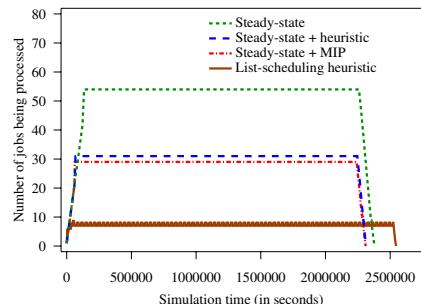
Figure 3(a) shows the performance of the four scheduling algorithms on a given scenario. The efficiency, that is the ratio of the optimal throughput, obtained by each algorithm is given for different batch sizes. The optimal throughput is easily obtained: it is the throughput that would obtain a steady-state schedule (as the one presented in [9]) on an infinite number of jobs.

The list-scheduling heuristic has a constant efficiency as soon as the size of the batch reaches a critical number of jobs (a few tens). On the opposite, the performance of the steady-state strategies evolves with the size of the batch, the more jobs to schedule, the more efficient are these strategies. With a very large size of batch, these strategies would all reach an efficiency of 100%, i.e., they would give the optimal steady-state throughput. In this study, we focus on batches with medium size, that is, when the number of jobs to schedule is large enough to concentrate on throughput maximization, but not extremely large or infinite as assumed in classical steady-state studies. On this particular example, all steady-state strategies achieve 90% of the optimal throughput as soon as there are 300 jobs to schedule.

Figures 4(a) and 4(b) display the proportion of scenarios where the algorithms reach 90% of the optimal throughput, depending on the size of the batch, both in the SIMPLE and GENERAL cases. We notice that the list-scheduling algorithm behavior does not depend on the batch size, and reaches a good performance (90% of the optimal throughput) only for 43% of the cases (in general). On the contrary, steady-state strategies give much better performance, reaching a good throughput in 60% of the cases for batches

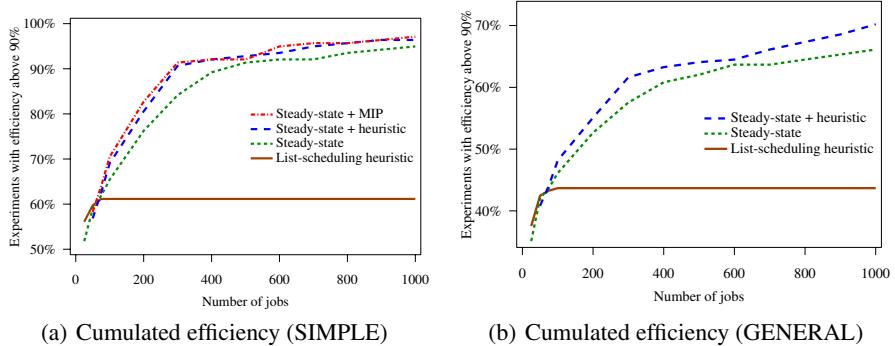


(a) Efficiency for batches of increasing size.



(b) Evolution of the number of running instances.

Fig. 3. Examples of results for efficiency and number of running instances

**Fig. 4.** Cumulated efficiency

with more than 400 jobs. Here, we are interested in comparing the performance of the different steady-state strategies. We notice that the performance of steady-state+MIP and steady-state+heuristic is better than steady-state: for medium-size batches, the number of jobs needed to get a good performance is smaller than in the original steady-state algorithm. This gap is noticeable even if it is not very large. In the SIMPLE case, (Figure 4(a)), we are able to compare steady-state+MIP with steady-state+heuristic: although the MIP strategy always gives better results, the heuristic performs very well, and the gap between both strategies is not always noticeable.

Our objective to reduce the number of dependencies seems to have a good correlation with makespan reduction. However there are some cases where reducing the number of dependencies leads to a worse makespan. This can be observed in figure 4(a) for batches of 500 jobs where the steady-state+heuristic performs better than the steady-state+MIP even if the steady-state+MIP resolves more dependencies. In general, resolving more dependencies result in less tasks to be executed in initialization and clean phases, that are known to be suboptimal. However, in a few cases, when the size of the batch, for particular size of batches, it may result in more tasks to be scheduled in those phases, thus leading to a worse makespan.

5.4 Number of Running Instances

Figures 3(b) presents the evolution of the number of running job instances on a given platform/application scenario. At a given time t , a running instance is a job which has been started (some tasks have been processed), but is not terminated at time t . Thus, temporary data for this instance have to be stored in some buffers of the platform. This figure illustrates the typical behavior of the steady-state algorithms. During the initialization phase, the number of job instances grows: instances are started to prepare for the next phase. During the steady-state phase, the number of job instances is roughly constant. Finally, in the termination phase, remaining instances are processed and the number of running instances drops to zero.

One of the drawbacks of steady-state scheduling presented in Section 2.3 is illustrated here: compared to another approach like list-scheduling, it induces a large number of running instances (on this example, 54 instead of 8). This example also shows that reducing the number of inter-period dependencies reduces the number of running instances for steady-state scheduling: with the MIP optimization, we get a maximum of 29 running instances, and 31 with the heuristic. We compared the maximum number of running instances in steady-state for the optimized versions (MIP and heuristic) and the original one: on average, steady-state+MIP induces a decrease of 35% and steady-state+heuristic reaches a decrease between 22% and 26% (respectively in the GENERAL and SIMPLE cases). Thus, our optimization makes steady-state scheduling more practical as it reduces the size of the required buffers.

5.5 Running Time of the Scheduling Algorithms

Figures 5(a) and 5(b) present the average time needed to compute the schedule of a batch depending on its size, in the SIMPLE and GENERAL cases. We first notice that the list-scheduling heuristic is extremely costly when the size of the batch is above a few hundreds. Its supra-linear behavior is due to the complexity of finding a ready task to schedule in a number of considered tasks that grows linearly in the size of the batch.

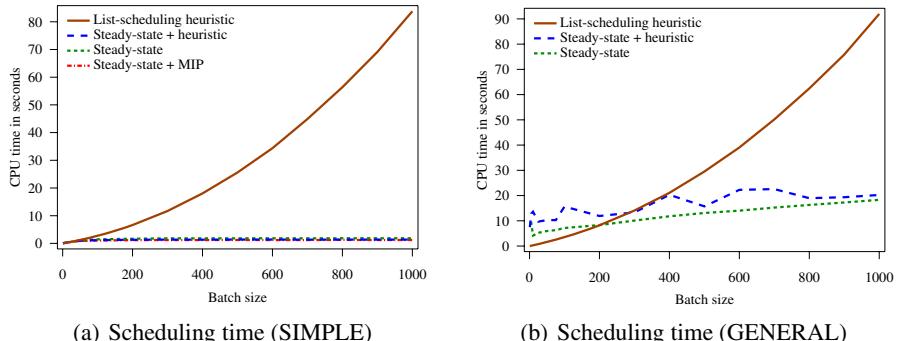


Fig. 5. Scheduling time in seconds

In the SIMPLE cases, the time needed to optimally solve the inter-period dependency minimization using the MIP is negligible, and the time needed to compute the periodic schedule is always below 2 seconds for all strategies. In the GENERAL cases, the period of the schedule is larger, and it induces more computation: initialization and termination phases are longer (and may increase with the size of the batch), thus the computation of their schedule takes some time. The optimization of the steady-state phase by the heuristic is also time-consuming. Anyway, the computation of the schedule with steady-state approaches never exceeds 20 seconds.

6 Conclusion

In this study, we have presented an adaptation of steady-state scheduling techniques for scheduling batches of task graphs in practical conditions, that is when the size of the batch is limited. The optimization we propose consists in a better usage of the period of the steady-state schedule. Instead of directly targeting the minimization of the makespan, we choose to reduce the number of inter-period dependencies. This problem is NP-complete, which justifies a solution based on Mixed Integer Programming. Our simulations show that this objective was relevant: when decreasing the number of inter-period dependencies, the throughput of the solution on medium-size batches is improved. Furthermore, the obtained solution requires less buffer space, since fewer instances are processed simultaneously, making the schedule even more practical. In future work, we plan to concentrate on small-size batches: since the optimal throughput is not reachable for these batches, it would be interesting to study non-conservative approaches, i.e., periodic schedules based on sub-optimal throughput, but which are more convenient to use thanks to their short period.

References

1. Foster, I.T., Kesselman, C. (eds.): *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, San Francisco (2004)
2. Chervenak, A., Foster, I., Kesselman, C., Salisbury, C., Tuecke, S.: The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of Network and Computer Applications* 23(3), 187–200 (2000)
3. Oinn, T.M., Addis, M., Ferris, J., Marvin, D., Senger, M., Greenwood, R.M., Carver, T., Glover, K., Pocock, M.R., Wipat, A., Li, P.: Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics* 20, 3045–3054 (2004)
4. Germain, C., Breton, V., Clarysse, P., Gaudeau, Y., Glatard, T., Jeannot, E., Legré, Y., Loomis, C., Magnin, I., Montagnat, J., Moureaux, J.M., Osorio, A., Pennec, X., Texier, R.: Grid-enabling medical image analysis. *Journal of Clinical Monitoring and Computing* 19, 339–349 (2005)
5. Lee, S., Cho, M.K., Jung, J.W., Weontae Lee, J.H.K.: Exploring protein fold space by secondary structure prediction using data distribution method on grid platform. *Bioinformatics* 20, 3500–3507 (2004)
6. Pitt-Francis, J., Garny, A., Gavaghan, D.: Enabling computer models of the heart for high-performance computers and the grid. *Philosophical Transactions of the Royal Society A* 364, 1501–1516 (2006)
7. Ludtke, S.J., Baldwin, P.R., Chiu, W.: EMAN: Semiautomated Software for High-Resolution Single-Particle Reconstructions. *Journal of Structural Biology* 128, 82–97 (1999)
8. Peng, L., Candan, K.S., Mayer, C., Chatha, K.S., Ryu, K.D.: Optimization of media processing workflows with adaptive operator behaviors. In: *Multimedia Tools and Applications. Computer Science*, vol. 33, pp. 245–272. Springer, Heidelberg (2007)
9. Beaumont, O., Legrand, A., Marchal, L., Robert, Y.: Steady-state scheduling on heterogeneous clusters. *Int. J. of Foundations of Computer Science* 16, 163–194 (2005)
10. Bertsimas, D., Gamarnik, D.: Asymptotically optimal algorithms for job shop scheduling and packet routing. *J. Algorithms* 33, 296–318 (1999)

11. Diakité, S., Nicod, J.M., Philippe, L.: Comparison of batch scheduling for identical multi-tasks jobs on heterogeneous platforms. In: PDP, pp. 374–378 (2008)
12. Diakité, S., Marchal, L., Nicod, J.M., Philippe, L.: Steady-state for batches of identical task graphs. Research report RR2009-18, LIP, ENS Lyon, France (2009)
13. ILOG: Cplex: High-performance software for mathematical programming and optimization (1997), <http://www.ilog.com/products/cplex/>
14. Topcuoglu, H., Hariri, S., Wu, M.-Y.: Task scheduling algorithms for heterogeneous processors. In: Proceedings of HCW 1999, Washington, DC, USA, p. 3. IEEE CS, Los Alamitos (1999)
15. Casanova, H., Legrand, A., Quinson, M.: SimGrid: a Generic Framework for Large-Scale Distributed Experiments. In: 10th IEEE International Conference on Computer Modeling and Simulation (2008)

A Buffer Space Optimal Solution for Re-establishing the Packet Order in a MPSoC Network Processor

Daniela Genius, Alix Munier Kordon, and Khouloud Zine el Abidine

Laboratoire LIP6, Université Pierre et Marie Curie, Paris, France

{daniela.genius, alix.munier, zineelabidine.khouloud}@lip6.fr

Abstract. We consider a multi-processor system-on-chip destined for streaming applications. An application is composed of one input and one output queue and in-between, several levels of identical tasks. Data arriving at the input are treated in parallel in an arbitrary order, but have to leave the system in the order of arrival. This scenario is particularly important in the context of telecommunication applications, where the duration of treatment depends on the packets' contents. We present an algorithm which re-establishes the packet order: packets are dropped if their earliness or lateness exceeds a limit previously fixed by experimentation; otherwise, they are stored in a buffer on the output side. Write operations to this buffer are random access, whereas read operations are in FIFO order. Our algorithm guarantees that no data is removed from the queue before it has been read. For a given throughput, we guarantee a minimum buffer size. We implemented our algorithm within the output coprocessor in the form of communicating finite state machines and validated it on a multi-processor telecommunication platform.

1 Introduction

The packet order in telecommunication applications depends strongly on each packet's content and is subject to important variations [1]. Well known in the networking domain under the name of *packet reordering*, the problem of out-of-order arrival of packets on the output side has been underestimated for a long time because the Transmission Control Protocol (TCP [2]) does not absolutely require in-order delivery. The performance of TCP however is severely penalized by useless re-sending of packets which arrive too late and are considered as lost. A detailed analysis [3] reveals that it is insufficient to rely on TCP's capabilities alone. Recently, the phenomenon has been studied experimentally to some extent, confirming its practical relevance [4].

For simplicity, in the following we call *packet re-ordering* the re-establishment of the order among packets that arrive at the output queue.

Telecommunication applications can be considered a category of streaming applications. Performance requirements of such applications can often only be met by mapping onto a Multi Processor System-on-Chip (MPSoC). A *task and*

communication graph (TCG), describing the application in the form of a set of coarse-grained parallel threads, can be of pipeline or of task farm flavor, or any combination of the two. However, a price has to be paid: data may be leaving the system in an order different from that in which they entered.

Disydent (Digital System Design Environment [5]), which we used to build an earlier platform [6] is based upon point-to-point Kahn channels [7]. While the original Kahn formalism is well suited for video and multimedia applications that can be modeled by a task graph where each communication channel has only one producer and one consumer, it is not convenient for telecommunication applications where several tasks access the same communication buffer. In [8] we generalize the Kahn model by describing applications in the form TCG where tasks communicate via multi-writer multi-reader (MWMR) channels, implemented as software FIFOs that can be accessed by any number of (hardware or software) reader and writer tasks. The access to such a channel is protected by a single lock. As any reader and any writer task can access the channel provided that it obtains the lock, the order of packets within a flow becomes completely arbitrary.

1.1 Example Application

The first step in the treatment of packet streams, called *classification*, ranges from the mere identification of the protocol (http, ftp, ...) to more in-depth analysis of the traffic type. Classification enables traffic management through e.g. the detection of illicit traffic (peer-to-peer). Apart from its great practical interest, it qualifies as an example because of its high degree of inherent parallelism and severe throughput requirements [9].

For a large majority of networking applications it is sufficient to consider only the *header* of a packet, which should consequently be stored in fast on-chip RAM. The *input task* accepts on-chip and off-chip addresses from two channels. It reads

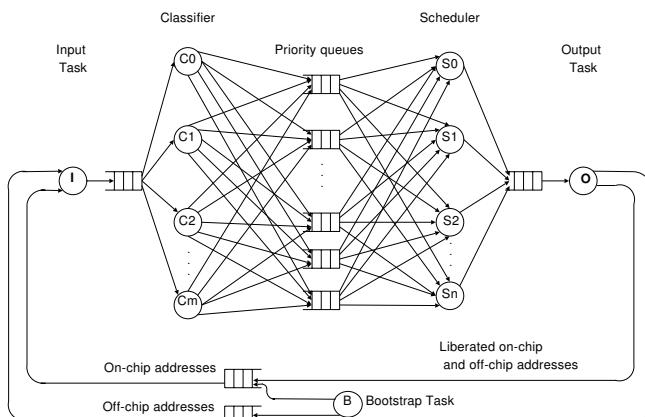


Fig. 1. Classification application task and communication graph

Ethernet encapsulated IP packets, cuts them into slots of equal size to fit the on-chip memory banks, and copies these slots to on-chip and off-chip memory, respectively. It produces a small packet *descriptor* containing the address of the first slot and additional information. A *classification task* reads a descriptor and then retrieves the packet from memory. The packet is deallocated if one of various checks fails. The classification task then writes the descriptor to one of several priority queues. The *scheduling tasks* ponder by the priority of the current queue and write the descriptor to the unique output queue if eligible. The *output task* constantly reads the output queue. Each time a slot is read and sent to the buffer, its liberated address is sent to either of the two channels for on-chip and off-chip addresses. A *bootstrap task* organizes the system startup.

1.2 Surface and Throughput Requirements

Memory takes up a large part of the silicon surface. This is particularly an issue for on-chip memory in embedded platforms. A hardware solution therefore has to cope with the problem of not impacting efficiency by algorithms too complex to implement while saving buffer space to temporarily store packets that are not yet allowed to leave.

In this paper we propose a buffer space-optimal packet order re-establishing algorithm for multimedia and telecommunication oriented MPSoC platforms. The algorithm numbers the incoming packets and enforces a strict order on the output side. Write operations to the re-ordering buffer are random access, whereas read operations are first-in first-out. Our hypothesis is that only very few data arrive excessively early or late; the latter are discarded. Our algorithm ensures that no data is removed from the buffer before it has been read. For a given throughput, we guarantee a minimum buffer size. We present an implementation within the output coprocessor in the form of communicating finite state machines and evaluate it experimentally.

Section 2 sums up the related work. Section 3 formulates the problem and states it quantitatively for a typical example. The algorithm itself is presented in Section 4. The final part of the paper describes the hardware implementation (Section 5) and confirms by experiment that the overall performance of the platform is preserved (Section 6). Finally, we point out directions of future work.

2 Related Work

From a purely algorithmic point of view, it is rather unusual to accept the loss of data; in consequence we did not find any similar work from that domain – in networking practice however, discarding of delayed packets is routinely done.

Commercial network processors often use a large number of smaller processors for straightforward packet treatment [1]. The packet order is then re-established by a central hardware component (see the combination of *Dispatch Unit* and *Completion Unit* of the IBM PowerNP series [10]). Buffers within this component are large in order to accommodate a wide variety of applications with higher or

lower degrees of disorder. Recent research tries to avoid a central control and use smaller buffers.

The out-of-order arrival of packets is closely related to the load balancing problem situated at the input side. Proposed solutions for the latter problem include adaptive load sharing for network multiprocessors [11] and software approaches like [12]. Evidently, those techniques cannot fully control the packet order on the output side. Among the re-ordering techniques situated on the output side, some accept a limited degree of disorder, others impose in-order delivery. An approach which imposes in-order delivery by combining buffering on input and output side is shown in [13]. This approach is based on timestamps (as opposed to sequence numbering) and results are derived for multi-stage Clos interconnects.

The *Reorder Density* function presented in [14] quantifies the cumulative degree of disorder in packet sequences. Like in our approach, packets are numbered sequentially at the input side and these numbers are compared to a receive index. Reorder Density is a cumulative metric applied to large traces of observed traffic (UDP or TCP), it is not employed to actively re-establish packet order in a given application; this requires more knowledge of the individual application.

3 Problem Formulation

In the simplest case, streaming applications can be considered as one input and one output task, with a treatment task in-between. When targeting very high throughput, many tasks running in parallel, one level of interconnect is generally insufficient to cope with the contention between various requests and responses. A multi-level architecture regroups processors around local interconnects. In such a NUMA (Non Uniform Memory Access) architecture, memory access times differ greatly depending on whether a processor accesses a memory bank local to the cluster, or on another cluster.

Let us now state the problem we wish to solve: packets leave the system in a different order than the one in which they entered it. Intuitively, this problem is best tackled from the output side: the output task has to ensure in-order delivery. Classically, only packets arriving too late are penalized (cf. the *time-to-live* counter in networking applications: packets are discarded if they pass too much time within the system). In order to transmit as large a number of packets as possible, we accept the discarding of a few which arrive excessively early.

Let us suppose that an infinite number of packets are to be reordered. At each time instant $i \in \mathbb{N}$, a packet arrives and is denoted by $\sigma(i)$. σ is clearly a bijective function from \mathbb{N} to \mathbb{N} . The packet arriving at time $i \in \mathbb{N}$ is on time if

Table 1. A sequence of packets σ

i	0	1	2	3	4	5	6	7	8
$\sigma(i)$	1	2	0	3	6	8	4	7	5

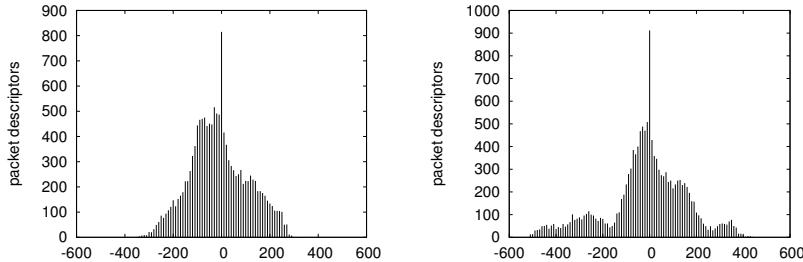


Fig. 2. Disorder for 10 (left) and 15 clusters (right) of 4 processors each

$\sigma(i) = i$. It is late (*Resp.early*) if $\sigma(i) < i$ (*Resp.σ(i) > i*). A (finite) example is presented by Table 1. Packet 7 is on time. Packet 4 is late and packet 8 is early.

Let us consider the introductory example of classification running on a generic clustered MPSoC featuring SoCLib [15] components based on the VCI shared memory paradigm and a two-level interconnect. Our platform contains a variable number of small programmable RISC processors (MIPS R3000). Application specific coprocessors are only used for the I/O of packet streams, they can be easily exchanged to treat other kinds of streams. Channels are all blocking; if an overflow occurs, packets are discarded from an internal buffer of the input coprocessor. As a consequence, no packet once numbered by the input coprocessor is lost.

Consider histograms of two configurations for flows of 54 byte packets (Figure 2). The x-axis shows the earliness/lateness of packets, the y-axis gives the number of packets with identical earliness/lateness. We represent earliness and lateness by calculating $i - \sigma(i)$ such that negative values represent earliness and positive values represent lateness. The graphs confirm our initial hypothesis that few packets arrive excessively early or late. We observe that the disorder is growing with the number of clusters.

4 Re-ordering Algorithm

The following strong hypotheses underlie our algorithm:

1. Packets are discarded if their earliness/lateness exceeds a given limit.
2. The throughput has to be preserved.
3. The buffer space required is minimal.

Our algorithm determines the minimal buffer size required to keep all packets whose earliness/lateness stays within the fixed limits. In the following we present our algorithm and prove its correctness.

Two integer values Δ_a and Δ_r are fixed and correspond respectively to the maximal earliness and the maximal lateness guaranteed, determined by experimentation as shown above. Every packet $\sigma(i)$ such that $\sigma(i) \in \{i - \Delta_r, \dots, i + \Delta_a\}$ must be reordered. Other packets may be ignored. For the previous example

with values $\Delta_a = 2$ and $\Delta_r = 2$, every packet except 5 and 8 must be reordered. So, the output of the reorder algorithm must be the sequence of integers from 0 to 8 without these two values. An array $\mathcal{S}[0 \cdots N - 1]$ of size N may be used to reorder the sequence with three main characteristics:

1. Values may be written in \mathcal{S} using random access memory. For any index $j \in \{0, \dots, N - 1\}$, a value may be written in $\mathcal{S}[j]$.
2. \mathcal{S} is emptied as a FIFO queue. Index $h \in \{0, \dots, N - 1\}$ is initialized to a given value h_0 . After each reading of $\mathcal{S}[h]$, the next value of h is $h + 1 \bmod N$.
3. Size N of \mathcal{S} must be minimum.

Theorem 1. *The size of \mathcal{S} must verify $N \geq \Delta_a + \Delta_r$.*

Proof. Let us consider the following sequence of $\Delta_a + \Delta_r$ packets: for every $i \in \{0, \dots, \Delta_r - 1\}$, $\sigma(i) = \Delta_a + i$. For every $i \in \{\Delta_r, \dots, \Delta_a + \Delta_r - 1\}$, $\sigma(i) = i - \Delta_r$. Clearly, the earliness of packets $\sigma(i)$ with $i \in \{0, \dots, \Delta_r - 1\}$ is Δ_a . The lateness of those for $i \in \{\Delta_r, \dots, \Delta_a + \Delta_r - 1\}$ is Δ_r . Thus all of them must be reordered.

At time Δ_r , all packets $\Delta_a + i$ for $i \in \{0, \dots, \Delta_r - 1\}$ must be yet stored in \mathcal{S} . Then, since \mathcal{S} is a FIFO queue for output, all other packets must be stored before, hence the result.

4.1 Algorithm

The size of the array \mathcal{S} is fixed to its minimum $N = \Delta_a + \Delta_r$. The idea is to store, at each step $i \in \mathbb{N}$, the value $\sigma(i)$ in $\mathcal{S}[\sigma(i) \bmod N]$ if $\sigma(i) \in \{i - \Delta_r, \dots, i + \Delta_a\}$ and to output the value stored (if any) in $\mathcal{S}[h_i]$ with $h_i = (i - \Delta_r) \bmod N$. Now, in order to optimize the use of \mathcal{S} , the output of $\mathcal{S}[h_i]$ must be done before the storage of $\sigma(i)$ in \mathcal{S} . This is always possible for $\sigma(i) \in \{i - \Delta_r + 1, \dots, i + \Delta_a\}$, but not for $\sigma(i) = i - \Delta_r$. In this case, $h_i = \sigma(i) \bmod N$ and the storage in \mathcal{S} must be done before; we call this the border condition. The algorithm in pseudo-code follows:

```

 $h := -\Delta_r \bmod N, i := 0$ 
While (true)
  If  $(\sigma(i) = i - \Delta_r)$  then
     $\mathcal{S}[h] := \sigma(i)$ 
    Output ( $\mathcal{S}[h]$ )
   $h := (h + 1) \bmod N$ 
  If  $\sigma(i) \in \{i - \Delta_r + 1, \dots, i + \Delta_a\}$  then
     $\mathcal{S}[\sigma(i) \bmod N] := \sigma(i)$ 
   $i := i + 1$ 
EndWhile

```

Table 2 presents an execution of the algorithm for the example presented by Table 1. The size of the intermediate buffer is fixed to $N = \Delta_a + \Delta_r = 4$.

Table 2. An execution for the sequence presented by Table 1. $\mathcal{S}[0 \dots 3]$ is the state of the intermediate array at the end of each iteration.

i	$\mathcal{S}[0]$	$\mathcal{S}[1]$	$\mathcal{S}[2]$	$\mathcal{S}[3]$	Output	Comments
0	.	1	.	.	.	
1	.	1	2	.	.	
2	.	1	2	.	0	
3	.	.	2	3	1	
4	.	.	6	3	2	
5	.	.	6	.	3	8 rejected
6	.	.	6	.	4	
7	.	.	6	7	.	
8	.	.	.	7	6	5 rejected
9	7	
10	

Before the loop, $h = 2$. For $i = 0$ and $i = 1$, $\sigma(0) = 1$ and $\sigma(1) = 2$ are stored in \mathcal{S} . Since $\mathcal{S}[2]$ and $\mathcal{S}[3]$ are empty for respectively $i = 0$ and $i = 1$, there is no output. For $i = 2$, $\sigma(2) = 0 = 2 - \Delta_r$ and $h = 0$. Thus, $\sigma(2)$ is stored in $\mathcal{S}[0]$ and directly sent to the output. For $i = 3$ and $i = 4$, output is respectively $\mathcal{S}[1]$ and $\mathcal{S}[2]$. $\sigma(3) = 3$ and $\sigma(4) = 6$ are stored also in $\mathcal{S}[3]$ and $\mathcal{S}[2]$. For $i = 5$, $\mathcal{S}[3]$ is sent to the output and $\sigma(5) = 8 < 5 + 2$ is rejected. For $i = 6$ and $i = 7$, $\mathcal{S}[0]$ and $\mathcal{S}[1]$ are both empty and there is no output. $\sigma(6) = 4$ and $\sigma(7) = 7$ are both stored in \mathcal{S} . For $i = 8$, $\sigma(8) = 5 < 8 - 2$ and thus, 5 is rejected. The buffer \mathcal{S} is then emptied.

The next theorem proves the correctness:

Theorem 2. *Every packet $\sigma(j)$ with $\sigma(j) \in \{j - \Delta_r, \dots, j + \Delta_a\}$ is ordered by the algorithm at iteration $\sigma(j) + \Delta_r$.*

Proof. By contradiction, let j be the smallest integer such that $\sigma(j) \in \{j - \Delta_r, \dots, j + \Delta_a\}$ is not ordered by the algorithm at iteration $i = \sigma(j) + \Delta_r$.

1. If $\sigma(j) = j - \Delta_r$, then $i = j$ and the value $\sigma(i)$ is sent at iteration j , a contradiction.
2. Let us assume now that $\sigma(j) \in \{j - \Delta_r + 1, \dots, j + \Delta_a\}$. At iteration i , $\mathcal{S}[h_i]$ does not contain $\sigma(j)$. Thus, $\sigma(j)$ was sent before or covered by another packet.
 - If $\sigma(j)$ was already sent, then it was at least at the iteration $i - (\Delta_r + \Delta_a) = \sigma(j) - \Delta_a$. But, $\sigma(j) \in \{j - \Delta_r + 1, \dots, j + \Delta_a\}$, thus $\sigma(j) - \Delta_a \leq j$. If $\sigma(j) - \Delta_a < j$, then $\sigma(j)$ was sent before being stored in \mathcal{S} , which is impossible. So, $\sigma(j) - \Delta_a = j$ and $\sigma(j)$ is sent at the iteration j . By the algorithm, we get that $\sigma(j) = j - \Delta_r$, a contradiction.
 - Let us suppose now that $\sigma(j)$ was covered by another packet $\sigma(k)$ before being sent by the algorithm. This value is sent at iteration $\sigma(j) + \Delta_r$, so $j < k \leq \sigma(j) + \Delta_r$. Since $\sigma(k)$ was stored, we get that $\sigma(k) \in \{k - \Delta_r, \dots, k + \Delta_a\}$. Lastly, since $\sigma(j)$ and $\sigma(k)$ were stored in the same place in \mathcal{S} , we get that $\sigma(j) \bmod N = \sigma(k) \bmod N$.

Let us prove that $\sigma(k) = \sigma(j) + N$. Indeed, since $\sigma(j) - \sigma(k) \leq j + \Delta_a - (k - \Delta_r) = (j - k) + N$, $j < k$ and $\sigma(j) \bmod N = \sigma(k) \bmod N$, we get that $\sigma(j) < \sigma(k)$. As $\sigma(k) \leq k + \Delta_a$ and $k \leq \sigma(j) + \Delta_r$, we get that $\sigma(k) \leq \sigma(j) + N$. As $\sigma(j) \bmod N = \sigma(k) \bmod N$, we conclude that $\sigma(k) = \sigma(j) + N$.

Now, since $\sigma(k) \leq k + \Delta_a$, we get $\sigma(j) + N \leq k + \Delta_a \leq \sigma(j) + \Delta_a + \Delta_r$ and thus $k = \sigma(j) + \Delta_r$. Now, as $\sigma(j) = \sigma(k) - N$, we obtain that $k = \sigma(k) - \Delta_a$, so $\sigma(k) = k + \Delta_a$.

Notice that $\sigma(k)$ is written in \mathcal{S} at the index $\sigma(k) \bmod N = (k - \Delta_r) \bmod N = h_k$. So, at the iteration k , $\mathcal{S}[h_k]$ is sent and then $\sigma(k)$ is written in $\mathcal{S}[h_k]$. So, $\sigma(k)$ can not cover any packet, a contradiction.

4.2 Extension

The algorithm may be easily extended if the numbering of the packets is not infinite. Indeed, let us suppose that the packets are numbered cyclically from 0 to $n - 1$ with $n \geq N$. Then, the packet numbered by 0 must be stored in \mathcal{S} just after the packet $n - 1$, thus we must have $n \bmod N = 0$. Now, under this assumption, the main modification of the algorithm remains in the acceptation test of $\sigma(i)$. Cyclic intervals defined as follows permit to modify it properly. Let x and y be two integers in $\{0, \dots, n - 1\}$. A cyclic interval from x to y is noted by $\{x, y\} \bmod n$ and is defined as: if $x \leq y$ then $\{x, y\} \bmod n = \{x, x + 1, \dots, y\}$. Otherwise, $\{x, y\} \bmod n = \{x, x + 1, \dots, n - 1, 0, \dots, y\}$. The acceptation test becomes $\sigma(i) \in \{i - \Delta_r + 1, i + \Delta_a\} \bmod n$.

5 Hardware Implementation

We implement the re-ordering algorithm within a cycle accurate bit accurate SystemC model of the output coprocessor. The sequence number $\sigma(i)$ is encoded in 20 bits of the packet descriptor produced by the input coprocessor. Only these descriptors transit the MWMR channels, implemented in shared memory [8]. The major difficulty now consists in the translation of the “pen and paper” algorithm into communicating automata which observe the SoCLib communication protocol VCI (Virtual Component Interconnect [16]).

The output coprocessor has one incoming VCI interface for chunks of packets and one incoming and two outgoing MWMR interfaces for packet descriptors and liberated on-chip and off-chip addresses, respectively. It re-assembles packets and re-establishes the packet order. The re-ordering buffer of configurable size M only needs to keep $\Delta_a + \Delta_r \leq M$ descriptors at a given time. In practice, Δ_a, Δ_r and M are powers of two. The re-ordering buffer is then read in a FIFO manner; write operations are random access at position $\sigma(i) \bmod N$, where N is the buffer size.

As the VCI protocol decouples requests to the interconnect from responses, two finite state machines have to be implemented which communicate with each other via signals and by interrogating each other’s state registers. The transitions of the simplified finite state machine in Figure 3b which sends the VCI

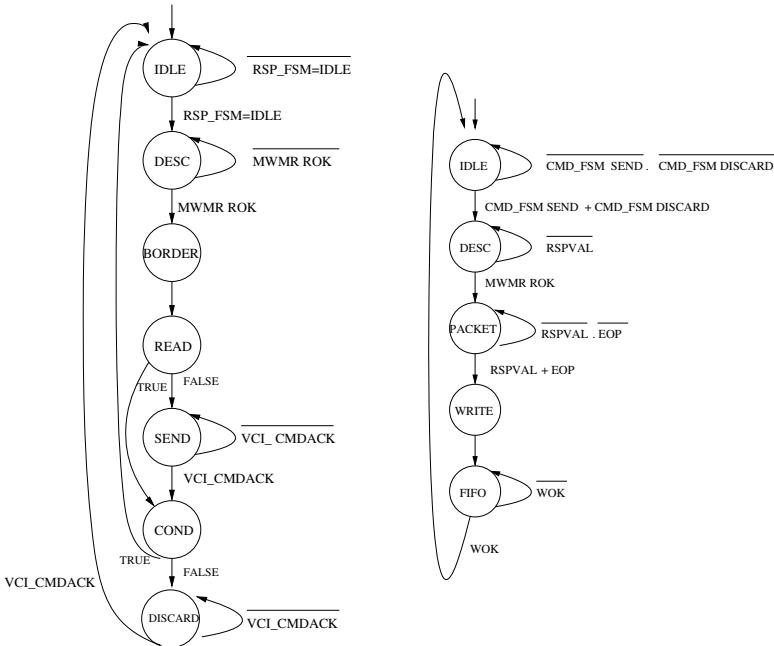


Fig. 3. (left) CMD FSM: VCI requests (right) RSP FSM: VCI responses

commands (thus CMD FSM as opposed to the RSP FSM for the VCI responses) reflect exactly the steps of our algorithm. A unique FSM reads the MWMR interface, retrieves the descriptor currently pointed by h , then either places it in the re-ordering buffer or discards it. In consequence, packets leave at the cadence of descriptor arrivals.

The CMD FSM checks the state register of the RSP FSM and starts working when this arrives in its IDLE state. It advances to the DESC state when the FIFO reading the incoming descriptor from the MWMR wrapper contains at least one packet descriptor. It then stores it in a register and goes to the BORDER state. If the border condition is satisfied, i.e. the packet lateness is $i - \Delta_r$, the descriptor is written to the re-ordering buffer \mathcal{S} at position h . Our algorithm assures that the re-ordering buffer can always be written. Only in the next state READ a descriptor is read from the buffer; this is either the one written in the BORDER state or one stored at position h in a previous COND state. The transition from the BORDER to the SEND state is unconditional. In the SEND state, if the read pointer h points to a valid descriptor, i.e. $\mathcal{S}[h]$ is not empty, the CMD FSM starts emitting the adequate commands in order to retrieve the packet through the VCI port and awaits the corresponding acknowledgments (VCI_CMDACK); the FSM iterates over all slots of a packet. We simplified this part in one state as the corresponding mechanisms have already been presented in earlier work. Otherwise, the FSM advances directly to the COND state where

it checks the general condition of the algorithm. The FSM decides either to keep or to discard the packet by testing whether the maximum earliness/lateness is comprised between $i - \Delta_r$ and $i + \Delta_a$.

If the packet is to be kept, its descriptor previously stored in the DESC state is read from its register and stored in the re-ordering buffer at position $\mathcal{S}[\sigma(i) \bmod N]$ and the FSM goes back to the IDLE state; else it goes on to the next state in order to deallocate the memory used by this packet (again, iterating over all slots of a packet, simplified by a single DISCARD state).

The RSP FSM (Figure 3b) is activated when the CMD FSM passes in either the SEND or the DISCARD state. It then accepts VCI requests (RSPVAL signal) for both kept and discarded packets, reconstitutes the packet by retrieving the slots from memory (PACKET state), writes them to a file or Ethernet (WRITE state) and in both cases sends liberated slot addresses to the corresponding channels for on-chip and off-chip addresses (FIFO state). The transition from the WRITE state to the FIFO state is unconditional.

Note that a single VCI interface retrieves both the packets which are kept and those which are discarded.

6 Test and Validation

We dimensioned our two-level clustered platform according to the results on number of processors, cache size, mapping obtained in [9]. The aim of our setup is to achieve a situation where both of the following conditions apply:

1. No packets are lost for a throughput imposed on the input side.
2. No packets are lost due to lack of space in the re-ordering buffer.

All ALU operations, including the expensive ones like modulo and integer divide, have to be simulated independently in order to derive a realistic measure of the slowdown of the output coprocessor. We connect input and output processor directly by a single MWMR channel. The re-ordering buffer within the output coprocessor was dimensioned at 2K words. These experiments (see Figure 4)

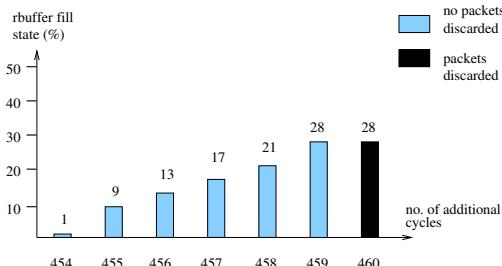


Fig. 4. Cost of the hardware implementation: cycles that can be added to simulate arithmetic operations without discarding packets. For an increasing number of cycles (x-axis), the fill state of the re-ordering buffer (y-axis) gradually increases.

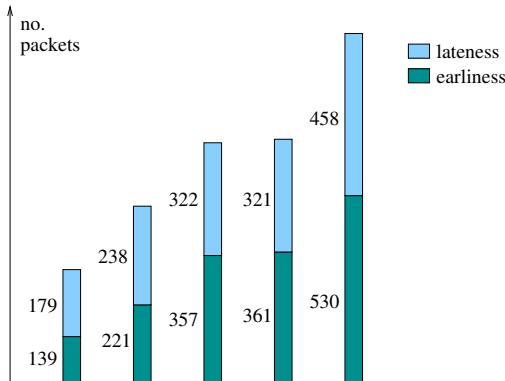


Fig. 5. Earliness and lateness: re-ordering buffer sizes for 6, 9, 10, 12 and 15 clusters

showed that for the current implementation of the output coprocessor, 459 clock cycles can be added before the throughput requirement of the input coprocessor is no longer matched by the output coprocessor (which means that the input coprocessor starts discarding packets); we add a waiting loop accordingly.

Figure 5 shows the impact of the number of clusters on the re-ordering buffer size. Our aim is that no packets are discarded due to excessive earliness or lateness. The minimum buffer size that allows re-ordering of packets where all packets are kept and retransmitted is the power of two superior to the sum of maximum earliness and lateness: 512 descriptors (4 Kbytes) in the first two cases, 1K descriptors (8Kbytes) otherwise. As can be seen, the buffer space has not only been proved minimal, it is quite small in practice. If chip surface and energy consumption impose stronger limitations, the buffer can be kept smaller. The number of packets discarded can then be determined experimentally.

7 Conclusion and Future Work

We propose an order re-establishing algorithm for a buffer of minimal size situated on the output side of a stream processing platform and show that the additional hardware does not significantly slow down the output coprocessor.

In the current version of the algorithm, packets leave the re-ordering buffer at the same cadence as they arrive; moreover each time an empty table entry is found, nothing is sent. Our mechanism is thus not work-conserving in the sense of [3] because it does not guarantee that a packet always leaves the system as soon as the outgoing Ethernet link becomes idle. An optimized version of the algorithm allows several contiguous cases of the re-ordering buffer to be liberated. It remains to be proved by implementation whether the increased complexity of implementation outweighs the improved potential throughput.

Our algorithm enforces a strict order, while per-flow re-ordering would be sufficient in many cases. It would moreover be interesting to compare statistics on the number of packets re-sent because they were discarded by our algorithm and

packets re-sent when using TCP without re-ordering in a real-world networking environment. It is straightforward to generalize our mechanism to other applications requiring in-order delivery while allowing to skip data, like for example video streaming.

References

1. Comer, D.: Network System Design using Network Processors. Prentice Hall, Englewood Cliffs (2003)
2. Postel, J.B., Garlick, L.L., Rom, R.: Transmission Control Protocol Specification. Technical report, Stanford Research Institution, Menlo Park (1976)
3. Bennett, J.C.R., Partridge, C., Sleetman, N.: Packet reordering is not pathological network behavior. *IEEE ACM Transactions on Networking* 7, 789–798 (1999)
4. Bellardo, J., Savage, S.: Measuring packet reordering. In: ACM SIGCOMM Internet Measurement Workshop, pp. 97–105. ACM Press, New York (2002)
5. Augé, I., Pétrot, F., Donnet, F., Gomez, P.: Platform-based design from parallel C specifications. *CAD of Integrated Circuits and Systems* 24, 1811–1826 (2005)
6. Berrayana, S., Faure, E., Genius, D., Pétrot, F.: Modular on-chip multiprocessor for routing applications. In: Danelutto, M., Vanneschi, M., Laforenza, D. (eds.) *Euro-Par 2004. LNCS*, vol. 3149, pp. 846–855. Springer, Heidelberg (2004)
7. Kahn, G.: The semantics of a simple language for parallel programming. In: Rosenfeld, J.L. (ed.) *Information Processing 1974: Proceedings of the IFIP Congress*, pp. 471–475. North-Holland, New York (1974)
8. Faure, E., Greiner, A., Genius, D.: A generic hardware/software communication mechanism for Multi-Processor System on Chip, targeting telecommunication applications. In: Proceedings of the ReCoSoC workshop, Montpellier, France (2006)
9. Genius, D., Faure, E., Pouillon, N.: Deploying a telecommunication application on a multiprocessor system-on-chip. In: Workshop on Design and Architectures for Signal and Image Processing, Grenoble, France (2007)
10. Allen, J.R., et al.: IBM PowerNP network processor: Hardware, software, and applications. *IBM Journal of Research and Development* 47, 177–193 (2003)
11. Kencl, L., Boudec, J.Y.L.: Adaptive load sharing for network processors. *IEEE ACM Transactions on Networking* 16, 293–306 (2008)
12. Chen, B., Morris, R.: Flexible control of parallelism in a multiprocessor PC router. In: USENIX Annual Technical Conference, Berkeley, CA, pp. 333–346 (2001)
13. Iyer, S., McKeown, N.: Making parallel packet switches practical. In: Proceedings of IEEE INFOCOM 2001, pp. 1680–1687. IEEE, Los Alamitos (2001)
14. Banka, T., Bare, A.A., Jayasumana, A.P.: Metrics for degree of reordering in packet sequences. In: LCN, pp. 333–342. IEEE Computer Society, Los Alamitos (2002)
15. SoCLib Consortium: The SoCLib project: An integrated system-on-chip modelling and simulation platform. Technical report, CNRS (2003), <http://www.soclib.fr>
16. VSI Alliance: Virtual Component Interface Standard (OCB 2.2.0). Technical report, VSI Alliance (2001)

Using Multicast Transfers in the Replica Migration Problem: Formulation and Scheduling Heuristics

Nikos Tziritas^{1,*}, Thanasis Loukopoulos², Petros Lampsas², and Spyros Lalidis¹

¹ Dept. of Computer and Communication Engineering, Univ. of Thessaly,
Glavani 37, 38221 Volos, Greece
{nitzirit,lalidis}@inf.uth.gr

² Dept. of Informatics and Computer Technology, Technological Educational Institute (TEI)
of Lamia, 3rd km. Old Ntl. Road Athens, 35100 Lamia, Greece
{luke,plam}@teilarlam.gr

Abstract. Performing replica migrations in the minimum possible time, also called the Replica Migration Problem (RMP), is crucial in distributed systems using replication. In this paper we tackle RMP when multicast transfers are available. We give a formal problem statement as a Mixed Integer Programming problem. It turns out that the problem is NP-hard. Therefore, we resolve to scheduling heuristics in order to find good solutions. Through simulations we identify different tradeoffs between performance and execution time and conclude on the most attractive approaches.

1 Introduction

Replication is widely used to achieve performance, availability and reliability in distributed data provision systems, e.g., web and video servers [1], [2]. Depending on the system, it may be necessary to occasionally change the employed *replica placement scheme*, i.e., the number of replicas per data object and the servers that should hold each replica. For instance, in distributed video servers this might be needed when new movies become available in anticipation of the increased demand for them. Of particular interest is how to optimally move from one replica placement scheme to the next, also called the *replica migration problem* (RMP).

The problem has been studied in the past in various environments. For example, [3] and [4] focus on disk farms, while [5] focuses on content distribution networks. More recently the problem has been incorporated in task scheduling over the Grid [7] where the aim is to minimize the makespan of task executions. In previous work [6], we investigated RMP in a generic distributed system assuming only point-to-point transfers. Here, we consider multicast-based transfers. The *multicast-based replica migration problem* (M-RMP) can be briefly stated as follows: given an initial and a target replica placement scheme, schedule a series of multicast-based transfers so that starting from the former the latter is obtained in the shortest possible time.

The main contributions of this paper are: (i) we motivate the use of multicast trees to obtain solutions for RMP; (ii) we discuss its relation to the Steiner-tree problem

* Supported in part by the Alexander S. Onassis Public Benefit Foundation in Greece.

[8]; (iii) we rigorously define M-RMP as a Mixed Integer Programming (MIP) problem; (iv) we propose heuristics based on the construction and scheduling of Steiner-trees; (v) we evaluate our heuristics using simulations and conclude on the tradeoffs between performance and running time. To the best of our knowledge this is the first paper tackling RMP with this scope.

2 Problem Description

Consider a distributed system consisting of M servers and N data objects. Let S_i denote the i^{th} server, $1 \leq i \leq M$, and O_k , $s(O_k)$ be the name and respectively the size (measured in abstract data units) of the k^{th} data object, $1 \leq k \leq N$. We say that S_i is a *replicator* of O_k if it has a replica thereof. Let X be a $M \times N$ replication matrix which encodes a replication scheme as follows: X_{ik} is 1 if S_i is a replicator of O_k , else it is 0.

Servers communicate via point-to-point bidirectional links. A link connecting S_i and S_j is denoted by l_{ij} and its capacity c_{ij} represents the number of data units that can be transferred over l_{ij} per time unit. Each object transfer is performed using one or more multicast trees, with each tree being rooted at a server that has a replica and transmits the object to one or more destination servers. The transfer rate is equal to the capacity of the bottleneck link, hence the duration of a multicast-based transfer is: *size of object/capacity of bottleneck link*. Notably, a multicast tree may involve intermediate servers that are not supposed to store a replica of the object being transmitted; these servers merely act as routers and do not store the object locally. The cost (in time) for routing and/or storing object data is considered to be negligible.

As an example, consider the system configuration shown in Fig. 1, and assume that server S_1 must send an object of size 8 to all other servers. One possible multicast tree for this transfer is depicted in Fig. 1(a) (bold lines). Given that the bottleneck links, l_{23} and l_{54} , have a capacity of 1, the transfer will complete in 8 time units. A better option is shown in Fig 1(b), where the transfer can be performed at a rate of 4, in 2 time units. This illustrates an important component of M-RMP, namely finding a *Steiner tree* that can support the highest transfer rate; or equivalently, the Steiner tree whose bottleneck link(s) have the highest capacity. We refer to it as the *maximum bandwidth Steiner-tree (MBST)*.

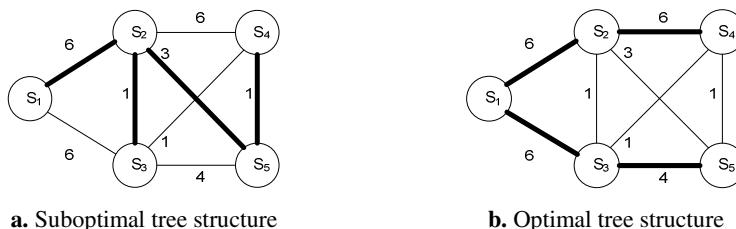


Fig. 1. A replication example with a single multicast tree

Now, suppose S_1 must send two objects of size 8 to all other servers. The problem now becomes to define and schedule several different MBSTs. In the special case where link capacities are unary, as shown in Fig. 2(a), the problem is equivalent to constructing disjoint Steiner trees, also known as *Steiner-tree packing problem* [9] which is NP-hard. However, the disjoint tree solution is not always optimal when link capacities are unequal. Fig. 2(b) depicts the same tree structures as Fig. 2(a), but assuming the link capacities of Fig. 1. In this case both trees have a bottleneck link with a capacity of 1, resulting in a total makespan of 8 time units. Fig. 2(c) shows the optimal multicast tree structure which involves (partially) overlapping trees. The black tree has a bottleneck capacity of 3 (l_{25}) and as a consequence can use links l_{12} , l_{24} and l_{13} at only half of their capacity. The remaining capacity can be used to schedule in parallel the grey tree, also at a rate of 3. Hence the makespan is 8/3 time units.

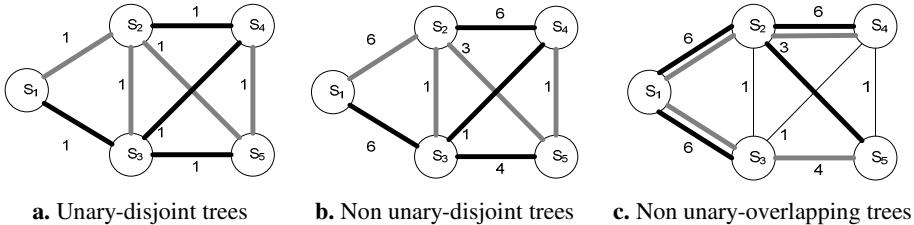


Fig. 2. A replication example with two multicast trees

M-RMP further differs from the classic Steiner-tree packing problem in two ways: (i) there may be multiple servers which could be used as sources (tree roots) for an object transfer; (ii) replicas created at one step could serve as additional sources for subsequent steps. In the following, we illustrate both aspects through examples. Notably, since the simplified case of the Steiner-tree packing problem is NP-hard, the more complicated M-RMP is NP-hard as well.

Consider the network of Fig. 3 where servers S_1 , S_2 , S_3 hold a replica of the same object which must be further replicated to the remaining servers. Clearly, the best solution is not to construct one but rather three separate multicast trees, one per source (shown in bold lines). The problem then extends to defining a *forest* of MBSTs, hence referred to as *maximum bandwidth Steiner-forest (MBSF)*. In Sec. 4 we discuss an algorithm to optimally construct a MBSF for replicating a single object.

Based on the same network topology but with link capacities and object sizes being unary, assume S_1 holds objects A and C, S_2 holds B and C, and S_3 holds A and B, as shown in Fig. 4. Suppose A and B must be replicated to the grey nodes, A and C to the white nodes, B and C to the black nodes, and all objects to the shaded nodes. One solution is to first schedule the entire forest for replicating A (shown in bold lines), then the forest for B, then the forest for C, in a sequential manner. This gives a makespan of 3. However, the best solution is to schedule individual trees of each forest. Specifically, in a first step, shown in Fig. 5(a), S_1 sends A to the grey nodes, S_2 sends C to the white nodes, and S_3 sends B to the black nodes. Shaded servers keep a copy of the objects that were transmitted through them. Then, in a second step, shown in Fig. 5(b), S_1 sends C to the shaded server below it, which (concurrently) sends A to

the white nodes, and so on for S_2 and S_3 , resulting in a total makespan of 2. The reason for the performance difference between these two schedules, is that in the first case only 2/3 of the nodes receive an object at each step, while at the second, every node does. Intuitively, it is tougher to involve a large number of nodes and links at each scheduling step when the schedulable unit is a whole forest as opposed to a tree. Both approaches are investigated via respective heuristics, described in Sec. 5.

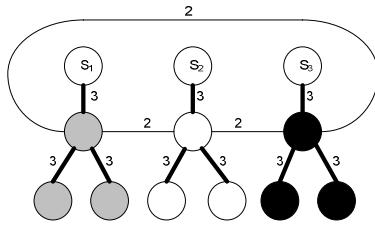


Fig. 3. An example of a Steiner forest

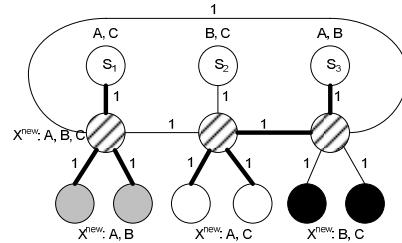
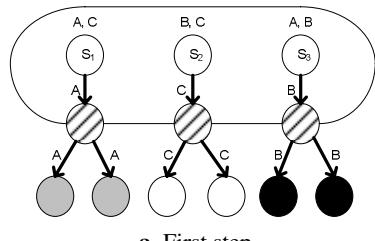
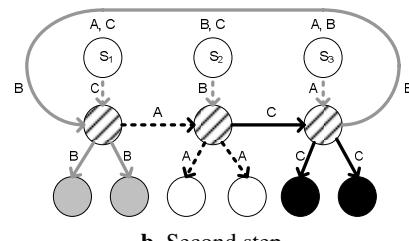


Fig. 4. An example requiring the use of intermediate sources to solve optimally



a. First step.



b. Second step.

Fig. 5. The optimal schedule of the example of Fig. 4

3 Mixed Integer Programming Problem Formulation

We formulate M-RMP as a MIP problem with constraints (1)-(12), some of them being quadratic. The idea is to consider a schedule of multicast tree object transfers and impose on it several validity requirements. Each transfer is modelled using a *transfer-start* and *transfer-end* event, associated with timestamps. In order to capture the tree-based multicast transfers and corresponding link reservations, a *dummy source server* (let S) and a *dummy destination server* (let D) is used, connected to all proper servers via *dummy links* of unlimited capacity. Each multicast tree has its root connected to S and all its destinations connected to D . The formulation also assumes a known upper bound for the number of events in the optimal schedule, let Z . An additional (dummy) void event at the end of the schedule indicates that the new replication scheme has been reached after the Z^{th} event. Thus, the total length of the schedule is $Z+1$, its tail comprised of at least one void event (or more, if the actual optimal schedule has fewer than Z events). A conservative bound for Z can be calculated by considering the extreme case where all tree transfers are point-to-point transfers:

$Z \leq 2 \times \text{outstanding replicas}$. Outstanding are the replicas which must be created in the new replication scheme. This number can be trivially determined as a function of X^{old} and X^{new} , by enumerating all elements for which holds: $X_{ik}^{\text{old}} = 0 \wedge X_{ik}^{\text{new}} = 1$.

Table 1 summarizes the variables used to describe the events in the schedule as well as additional problem variables. Unless otherwise stated, the indices in the variables take the following values: $1 \leq k \leq N$, $1 \leq i, j \leq M$, $1 \leq u \leq Z$, $1 \leq v < Z$. The Replica Migration Problem can then be stated as: *minimize $\ell^{\overline{Z}+1}$ subject to constraints (1)-(12).*

Constraints (1)-(5) relate event types with each other as well as with the state of the replication matrix. Specifically, (1) states that each event is a multicast transfer-start, a multicast transfer-end, or void, and that the last event is void. (2) requires that the replication process starts with replication matrix X^{old} and reaches X^{new} . (3) captures the fact that if the u^{th} event is a transfer-end for O_k then for all destination servers S_i (connected to D), the entry X_{ik} in the subsequent replication matrix is equal to 1. (4) gives the preconditions for a transfer, namely, in order for a transfer to start, the source server must have a copy of the object and the destination servers must not. (5) states that each transfer-start event has a corresponding transfer-end event which occurs later in the schedule, and that the number of transfer-end events must be equal to the number of transfer-start events (no orphan events are allowed).

Constraints (6)-(9) capture the fact that each multicast transfer is done using a tree structure. (6) states that for each multicast transfer each server has at most one incoming link (parent), and that the number of its outgoing links are at least equal to the number of incoming (recall that destinations are connected to D). (7) ensures that the tree is connected by requiring that no server exists which has outgoing links without having any incoming links. (8) captures the absence of cycles, requiring that the relative depth of a child node is greater than the depth of its father (the precise values do not matter). (9) states that S is a virtual parent for the tree root of each transfer.

Constraints (10)-(11) capture link-bandwidth reservations. (10) demands that corresponding transfer-start and transfer-end events relate to the same links. (11) requests that the aggregate rates of ongoing transfers do not exceed link capacity.

The final set of constraints (12) keeps track of time. The first part states that events must be properly ordered in time, while the second part states that the timestamp of a transfer-end event is equal to the sum of the transfer-start timestamp and the transfer duration. Note that the rate of a multicast transfer increases when capacity is freed at its bottleneck link(s) due to the completion of other transfers. Thus its duration is computed by subtracting from the object size the amount of data transmitted during each interval where the rate of the multicast has been increased, and dividing the remaining object size with the transfer rate that holds until its completion.

In terms of complexity, the number of variables and constraints is roughly $O(M^3N^3)$ and $O(M^4N^2 + M^3N^3)$, respectively, for M servers, N objects and assuming that Z is $O(MN)$ in the worst case and that the number of links is $O(M^2)$ for relatively dense graphs. We have implemented this MIP problem in a commercial optimizer [10]. Even though optimizers can tackle linear programming problems of this size, the fact that (some) constraints are quadratic limits scalability. As a consequence, we were able to obtain solutions, within acceptable time, only for small problem sets (5 objects and servers). This did not allow us to make satisfactory comparisons with the heuristics described in the next sections, which were evaluated for considerably bigger problem sets.

Table 1. Problem variables

Variable	Description
T_{ik}^u	1 iff the u^{th} event is a transfer-start for O_k , through a tree rooted at S_i , else 0
E_{ik}^{uv}	1 iff the u^{th} event is a transfer-end event for the start event T_{ik}^v , else 0
V^u	1 iff the u^{th} event is void (voids occur at the end of the schedule), else 0
X_{ik}^u	1 iff S_i has a replica of O_k before the u^{th} event occurs, else 0
l_{ij}^u	1 iff the u^{th} event is a transfer start/end and l_{ij} belongs to the tree, else 0
d_i^u	the relative depth of server S_i in the tree formed at the u^{th} event
t^u	the time when the u^{th} event occurs (real ≥ 0)
r^{uv}	the rate of the transfer started at the v^{th} event between t^u and t^{u+1} (integer $\neq 0$)

$$\sum_{\forall i} \sum_{\forall k} T_{ik}^u + \sum_{\forall i} \sum_{\forall k} \sum_{\forall v} E_{ik}^{uv} + V^u = 1 \quad \forall u, \quad V^{Z+1} = 1. \quad (1)$$

$$X_{ik}^1 = X_{ik}^{\text{old}} \quad \forall i, k, \quad X_{ik}^{Z+1} = X_{ik}^{\text{new}} \quad \forall i, k. \quad (2)$$

$$X_{ik}^{u+1} = X_{ik}^u + \sum_{\forall j} \sum_{\forall v} E_{jk}^{uv} l_{jD}^v \quad \forall i, k, u. \quad (3)$$

$$T_{ik}^u \leq X_{ik}^u \quad \forall i, k, u, \quad \sum_{\forall i} T_{ik}^u l_{jD}^u \leq 1 - X_{jk}^u \quad \forall k, j, u. \quad (4)$$

$$T_{ik}^u = \sum_{v > u} E T_{ik}^{vu} \quad \forall i, k, u, \quad \sum_{\forall i} \sum_{\forall k} \sum_{\forall u} T_{ik}^u = \sum_{\forall i} \sum_{\forall k} \sum_{\forall u} \sum_{\forall v} E_{ik}^{uv}. \quad (5)$$

$$\sum_{\forall i} l_{ij}^u \leq 1 \quad \forall j, u, \quad \sum_{\forall i} l_{ij}^u \leq \sum_{\forall x} l_{jx}^u \quad \forall j, u. \quad (6)$$

$$(1 - \sum_{\forall i} l_{ij}^u) \sum_{\forall x} l_{jx}^u = 0 \quad \forall j, u. \quad (7)$$

$$d_i^u l_{ij}^u < d_j^u \quad \forall i, j, u. \quad (8)$$

$$l_{Si}^u = \sum_{\forall k} T_{ik}^u + \sum_{\forall k} \sum_{\forall v < u} E_{ik}^{uv} \quad \forall i, u. \quad (9)$$

$$(l_{xy}^u - l_{xy}^v) \leq 1 - \sum_{\forall i} \sum_{\forall k} E_{ik}^{uv} \quad \forall x, y, u, v. \quad (10)$$

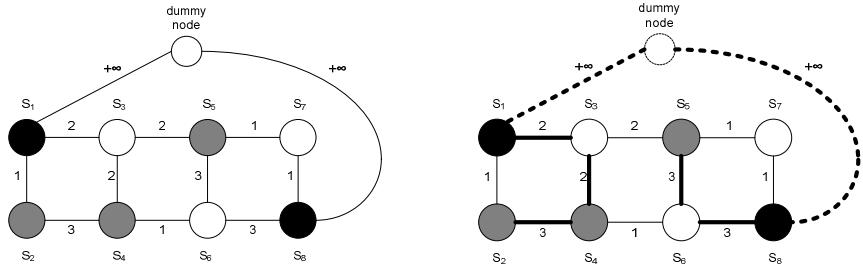
$$(l_{ij}^v + l_{ji}^v) \sum_{\forall v < u} r^{uv} \leq c_{ij} \quad \forall i, j, u. \quad (11)$$

$$t^u \leq t^{u+1} \quad \forall u, \quad (t^u - t^{u-1} - \frac{s(O_k) - \sum_{\forall v < x < u} (t^x - t^{x-1}) r^{(x-1)v}}{r^{(u-1)v}}) E_{ik}^{uv} = 0 \quad \forall i, k, 1 < u \leq Z, v. \quad (12)$$

4 Maximum Bandwidth Steiner-Forest Construction

As we have pointed out, the construction of maximum bandwidth Steiner-forests (MBSFs) is central to M-RMP. In this section, we present an algorithm for constructing a maximum bandwidth Steiner-tree (MBST) optimally, and discuss how it can be used to build a maximum bandwidth Steiner-forest (MBSF).

Given a network topology and a subset of nodes, the MBST problem is to define a tree connecting a single source node to all destination nodes, such that the minimum end-to-end capacity across its links is maximized. Although the basic Steiner-tree problem statement is NP-hard, it turns out that MBST can be solved optimally by the following simple algorithm: (i) find the maximum bandwidth path between the source and each destination (which can be achieved using a modified version of Dijkstra's algorithm; (ii) merge the previously defined paths to produce the MBST. Notice that optimality is due to the fact that the individual paths defined by Dijkstra are optimal and can be merged as is, since they are used to transfer the same object.



a. Network with dummy node: black nodes are replicators, grey nodes are destinations

b. The obtained Steiner forest (2 trees) after constructing the MBST for the dummy node

Fig. 6. An example of constructing a maximum bandwidth Steiner forest

To calculate an MBSF for a given object we augment the network by adding a *dummy node* and connecting it using *dummy links* with *infinite capacity* to all replicators of that object, as shown in Fig. 6(a). Then, we apply the MBST algorithm with the dummy node as the root, and discard the dummy node and edges to obtain the forest, as shown in Fig. 6(b). Each tree may support a different throughput depending on the capacity of its bottleneck link(s), e.g., 2 and 3 for the two trees in Fig 6(b).

The above algorithm produces the optimal MBSF. In other words, for the replication of a single object M-RMP can be solved to optimality regardless of the number of replicators and network parameters. For multiple objects M-RMP is NP-hard, having as a component the Steiner-tree packing problem [9], as discussed in Sec. 2.

5 Scheduling Heuristics

The algorithms presented here work iteratively. In each iteration, one or more objects are chosen and their corresponding (target) replication schemes are implemented, either partially or completely. This is done by producing an MBSF for each object, as

discussed in Sec. 4. The algorithms differ on how often they compute a forest for each object and the selection criteria used to decide which forests (or individual trees) to schedule next.

Prefixed Forest Algorithm (PFA). This algorithm works based on prefixed forests. An optimal MBSF is calculated for each object, once, assuming an empty network. Starting from an empty network, PFA picks a forest randomly and schedules it, i.e., updates the available link capacities. It then checks whether any of the remaining forests can be accommodated, picks/schedules one randomly, and the process is repeated. A forest cannot be chosen if it requires one or more links that are already fully utilized. When no more forests can be accommodated, the algorithm checks the earliest point in time when a scheduled multicast transfer (tree) finishes. If the link capacity that becomes available enables the scheduling one of the remaining forests, the above process is repeated. In any case, the next earliest point in time when the next multicast transfer finishes is checked, etc, until all forests are scheduled.

Earliest Start Forest Algorithm (ESFA). PFA attempts to schedule the prefixed forests that were calculated based on an empty network. This creates a potential limitation as shown in Fig. 7. Assume that the black node holds two objects of size 6 which must be replicated on the white nodes. PFA will define two identical forests (in fact, trees) for the objects, shown with bold lines in Fig. 7(a). The two trees will be scheduled sequentially and the total makespan will be 4 time units. Notice, however, that after scheduling the first tree the remaining links form an alternative multicast tree, shown with dashed lines in Fig. 7(b), which can be employed to transfer the second object, concurrently to the first one, giving a total makespan of 3. The example demonstrates that it might be advantageous to schedule object transfers using suboptimal trees if these can be scheduled earlier than their “optimal” counterparts. This is the intuition behind ESFA. Specifically, ESFA works in a similar way to PFA, but *at each step re-computes* the forests for all pending (not yet replicated) objects based on the available capacity of the links.



Fig. 7. An example of PFA and ESFA executions

Earliest Completion Forest Algorithm (ECFA). ESFA tends to schedule the replication of an object earlier, utilizing more links compared to PFA. However, this is not always beneficial. Consider the example of Fig. 7 and assume that the links of capacity 2 had a capacity of 1 instead. The schedule produced by ESFA would involve the same two trees of Fig. 7(b), but the total makespan would now be 6; worse than the makespan of the PFA schedule which is 4. ECFA attempts to overcome this drawback by computing the best forests in terms of *completion time*, given both the *current and future* link states, based on the transfers already scheduled. For a single object this involves calculating an

MBSF in *all distinct points in the future* where a scheduled multicast (tree) finishes and bandwidth is released. Returning to the example of Fig. 7, this means that after the forest for the first object is scheduled (bold lines), the algorithm will calculate two forests for the second object, one for time 0 and one for time 2 when the first scheduled multicast tree finishes. Among the two options it will select the forest leading to earliest completion time. In the general case where multiple objects are involved, ECFA works in iterations, calculating for each object (whose replication has not been scheduled) the best forest in terms of completion time and adds it to a candidate list. Among the candidates, the one starting earlier is selected (ties are broken randomly).

Per tree scheduling. The previous algorithms schedule at the granularity of *entire* forests, i.e., in each step the full replication scheme of an object is scheduled to start at the same point in time. Another possibility is to consider scheduling at the granularity of individual trees. Intuitively, by scheduling trees instead of forests it should be possible to achieve better link utilization per scheduling step. To differentiate between the forest-based and the tree-based scheduling modes, we refer to the latter algorithm versions as PTA, ESTA, and ECTA, respectively. Notice that the tree-based versions may schedule the replication of an object in different chunks, each multicast starting at a different point in time. As a consequence the forest computed for each object in each step only needs to consider the *remaining* destinations according to the target replication scheme. Also, forests computed later in the schedule may exploit as sources *new* replicas which may have been created *in the meantime*.

6 Experiments

We have evaluated the algorithms presented in Sec. 5 using simulated experiments for different system configurations and replication scenarios. The server network was generated using the BRITE tool [11], for 35 nodes. We experimented with 4 different graph types with node connectivity of 1 (tree network), 2, 3 and 4, respectively. For each type, we generated 5 different network topologies which were used to do all measurements, averaging the results. Node connections followed the Barabasi-Albert model, which has been used to describe power-law router graphs [12]. Links were assigned a fixed transfer capacity, uniformly distributed between 1 and 10. A set of 100 objects was used, their size varying uniformly between 100 and 1000 data units.

6.1 Evaluating Forest-Based Scheduling Algorithms

In this section we compare the performance of PFA, ESFA and ECFA and identify the most promising algorithms. For this purpose, we let X^{old} feature a single (randomly placed) replica per object while X^{new} features additional replicas per object, varying uniformly in the ranges 2..5, 2..15 and 2..34. Note that since there is only one initial replicator per object, the forest constructed for it consists of one multicast tree.

Fig. 8 shows the makespan achieved by the forest-based algorithms for network graphs with connectivity 2. It can be seen that the makespan of all algorithms increases as more replicas need to be created per object. This is expected since with more nodes participating in a forest, the potential for scheduling many forests/trees in parallel decreases, as illustrated in Fig. 9. PFA has a clearly inferior makespan because it works with precomputed forests, thus missing parallelization opportunities. Interestingly,

ESFA slightly outperforms ECFA, managing to schedule more trees in parallel even if each tree is likely to support a lower throughput compared to ECFA.

In terms of execution time, PFA is about an order of magnitude faster compared to ESFA which in turn is an order of magnitude faster than ECFA, as shown in Fig. 10. To understand why, recall that PFA *never* recomputes a forest, whereas ESFA recomputes the forests of outstanding objects *only* for the earliest possible start time, and ECFA recomputes forests for *all possible* start times.

To further evaluate the above findings we repeated the same experiments for all 4 types of networks (connectivity 1, 2, 3 and 4), with the additional number of replicas per object varying uniformly in the range 2..15. The results are shown in Fig. 11. It can be seen that: (i) the makespan drops as the node connectivity increases, drastically between connectivity 1 and 2, more smoothly thereafter; (ii) for connectivity 2, 3 and 4, PFA produces worse schedules compared to ESFA and ECFA (which are comparable in all cases). Notably, networks with connectivity 1 are *trees*, in which case, all algorithms will compute the *same* forest (actually a tree) for the replication of each object. Also, a tree-formed network considerably reduces the degree of parallelism, for which increased opportunity arises in networks with larger connectivity. The runime of the algorithms followed the trend already shown in Fig. 10.

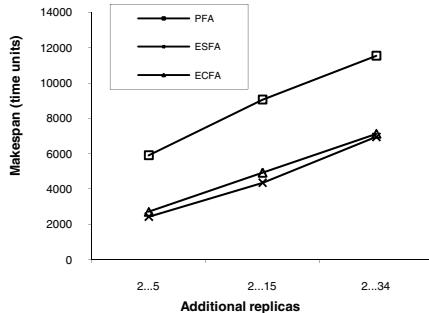


Fig. 8. Makespan vs additional replicas (connectivity: 2, initial replicas: 1)

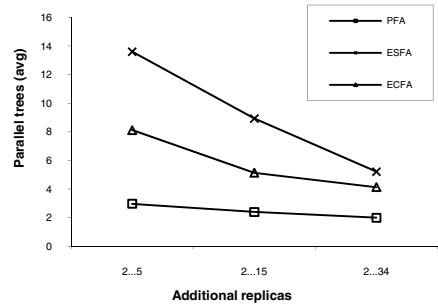


Fig. 9. Average number of trees per time unit (connectivity: 2, initial replicas: 1)

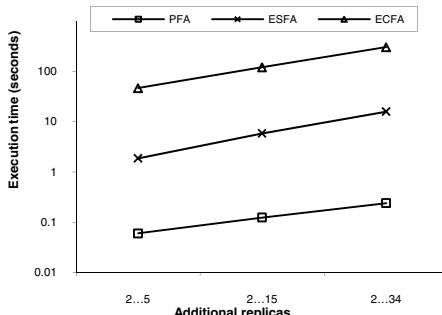


Fig. 10. Execution time vs additional replicas (connectivity: 2, initial replicas: 1)

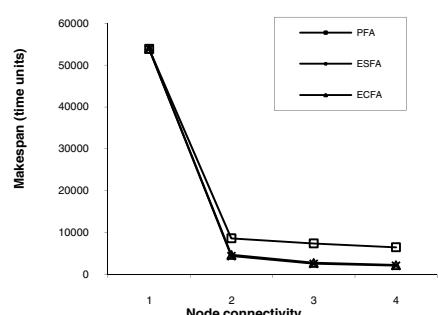


Fig. 11. Makespan vs connectivity (initial replicas: 1, additional replicas: 2..15)

6.2 Forest-Based vs. Tree-Based Scheduling Algorithms

In this section we compare ESFA and ECFA versus their tree-based alternatives ESTA and ECTA. We omit PFA and PTA because their results were considerably worse compared to the other algorithms (along the lines of the previous comparison).

To evaluate the ability of the algorithms to exploit different sources, we repeat the experiment of Fig. 11, but with 3 (instead of just 1) initial replicators. Fig. 12 shows the makespan for networks with connectivity 2, 3 and 4. As it can be seen, ECTA consistently outperforms ECFA in terms of makespan by 10% up to 20%. ESTA is slightly worse than ESFA for connectivity 2, but becomes increasingly better for connectivity 3 and 4. The generally improved results of the tree-based algorithms are due to the fact that with 3 initial replicators each forest may consist of several (up to 3) trees, giving rise to more opportunities for parallelism if trees are considered separately. Also, both ESTA and ECTA better exploit the increased parallelization potential for networks with larger connectivity, as confirmed in Fig. 13 which plots the average number of parallel multicast trees in the schedules produced. For completeness, we note that the observed trends also hold for connectivity 1 (not shown here).

Finally, the performance of the algorithms is investigated for connectivity 2, varying the number of initial replicators per object while keeping the number of additional number of replicas to be created per object the same as in the previous experiment (varying uniformly in the range 2..15). Fig. 14 plots the makespan of the respective schedules. Once again, ECTA consistently outperforms all other algorithms except for the case with only 1 initial replicator per object (where each forest is guaranteed to comprise of a single tree). In fact, the difference between ECTA and the rest increases to the number of initial replicators. This is expected for ECFA/ESFA because they tend to block until an entire forest becomes schedulable. However, this is less obvious for ESTA. Recall that ESTA favors trees with the earliest start time as opposed to the ones with the earliest completion time chosen by ECTA. If a path that is not (fully) utilized exists between a destination and the source of a tree, ESTA will include this node to the tree. Consequently the trees selected by ESTA are likely to (i) have more nodes, (ii) span across more links, and (iii) support a lower transmission rate compared to those of ECTA. This is partly confirmed in Fig. 15 which plots the average tree height of the schedules produced (tall trees are more likely to have a larger number of nodes, involve more links and support a lower transmission rate, compared to short ones). Also note that (ii) most likely reduces the degree of parallelism. Moreover, the schedules produced by ESTA have fewer opportunities to exploit newly created replicas as roots for subsequent multicasts, as shown in Fig. 16 (the forest-based algorithms are not shown, since, by construction, they use only initial replicators as roots of their multicast trees). This is because in ESTA the transfer of most outstanding object replicas will start early on in the schedule (where only initial replicators exist) and finish too late (due to the lower transfer rate) for the new replicas to be used in subsequent transfers. The improved makespan of the tree-based algorithms comes at the price of a longer execution time compared to the forest versions, as shown in Fig. 17.

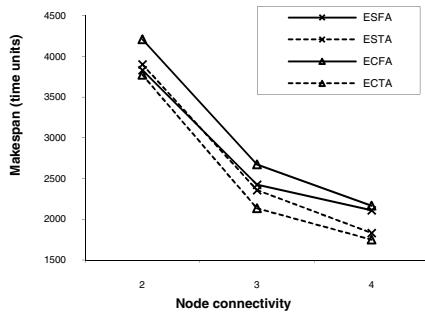


Fig. 12. Makespan vs connectivity (initial replicas: 3, additional replicas: 2...15)

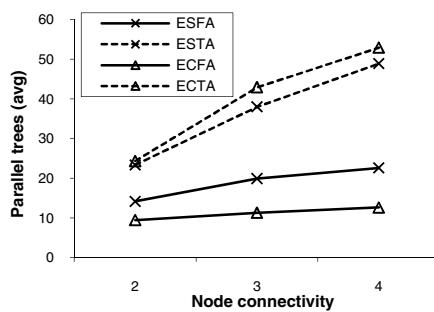


Fig. 13. Average number of parallel trees (initial replicas: 3, additional replicas: 2...15)

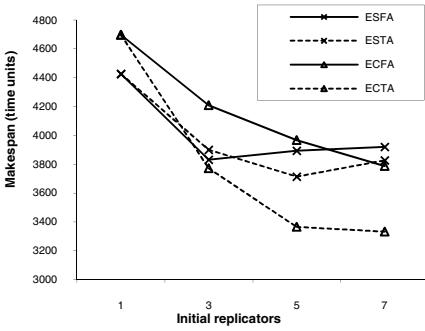


Fig. 14. Makespan vs initial replicas (connectivity: 2, additional replicas: 2...15)

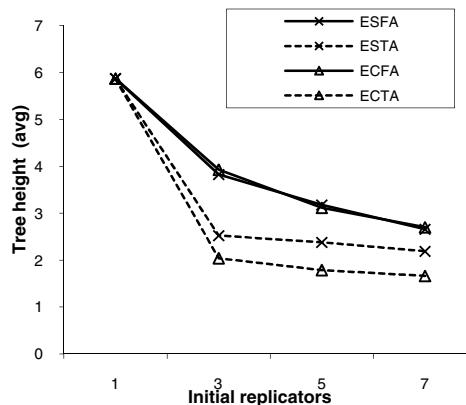


Fig. 15. Average tree height vs initial replicas (connectivity: 2, additional replicas: 2...15)

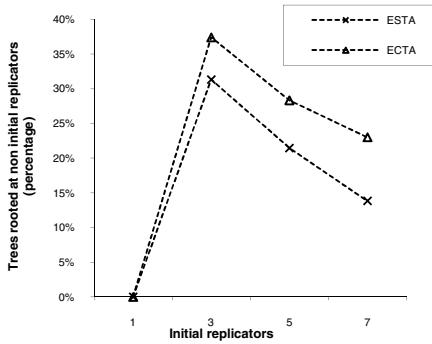


Fig. 16. Trees rooted at a new source (connectivity: 2, additional replicas: 2...15)

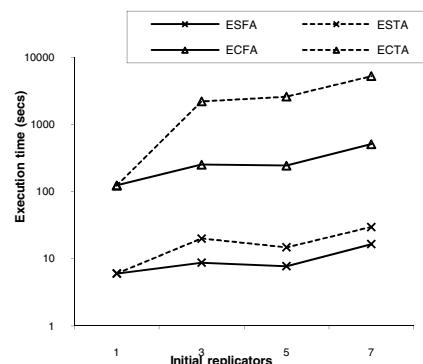


Fig. 17. Execution time vs initial replicas (connectivity: 2, additional replicas: 2...15)

7 Conclusions

In this paper we tackled the Replica Migration Problem from the standpoint of using one or more multicast trees with potentially different data transmission rates to implement the migration from the old replication scheme to the new one. The problem was rigorously captured as a Mixed Integer Programming formulation, and indicative heuristics were developed and evaluated, indicating different tradeoffs between makespan of the generated schedules and execution time of the algorithms. Summarizing our findings we can state that: (i) if execution time is the primary concern or the network is a tree and there is a single initial replica, PFA (and PTA) are the best choices; (ii) if the key objective is to achieve a short makespan, ECTA is the candidate producing significantly better schedules, especially for configurations with a large connectivity and/or several initial replicas. As part of our future work we plan to extend the experimental evaluation, as well as explore the case where servers have limited storage and routing capacity.

References

1. Khan, S., Ahmad, I.: Comparison and Analysis of Ten Static Heuristics-Based In-ternet Data Replication Techniques. *JPDC* 68(2), 113–136 (2008)
2. Laoutaris, N., Smaragdakis, G., Oikonomou, K., Stavrakakis, I., Bestavros, A.: Distributed Placement of Service Facilities in Large-Scale Networks. In: Proc. INFOCOM 2007, pp. 2144–2152.
3. Hall, J., Hartline, J., Karlin, A., Saia, J., Wilkes, J.: On Algorithms for Efficient Data Migration. In: Proc. SODA 2001, pp. 620–629 (2001)
4. Khuller, S., Kim, Y.A., Wan, Y.C.: Algorithms for Data Migration with Cloning. In: Proc. PODS 2004, pp. 448–461 (2004)
5. Killian, C., Vrable, M., Snoeren, A., Vahdat, A., Pasquale, J.: Brief Announcement: The Overlay Network Content Distribution Problem. In: Proc. PODC 2005, p. 98 (2005)
6. Tziritas, N., Loukopoulos, T., Lampsas, P., Lalis, S.: Formal Model and Scheduling Heuristics for the Replica Migration Problem. In: Luque, E., Margalef, T., Benítez, D. (eds.) *Euro-Par 2008. LNCS*, vol. 5168, pp. 305–314. Springer, Heidelberg (2008)
7. Desprez, F., Vernois, A.: Simultaneous Scheduling of Replication and Computation for Data-Intensive Applications on the Grid. *J. Grid Computing* 4(1), 19–31 (2006)
8. Wu, B.Y., Chao, K.M.: *Spanning Trees and Optimization Problems*, ch. 7. Chapman & Hall/CRC, Boca Raton (2004)
9. Lau, L.C.: An Approximate Max-Steiner-Tree-Packing Min-Steiner-Cut Theorem. In: Proc. FOCS 2004, pp. 61–70 (2004)
10. <http://www.lindo.com>
11. Medina, A., Lakhina, A., Matta, I., Byers, J.: BRITE: Boston University Representa-tive Internet Topology Generator (March 2001),
<http://cs-pub.bu.edu/brite/index.htm>
12. Barabasi, A.L., Albert, R.: Emergence of Scaling in Random Networks. *Science* 286, 509–512 (1999)

A New Genetic Algorithm for Scheduling for Large Communication Delays

Johnatan E. Pecero¹, Denis Trystram¹, and Albert Y. Zomaya²

¹ LIG Grenoble University, 38330 Montbonnot Saint-Martin, France

² The University of Sydney, Sydney, NSW 2006 Australia

Abstract. In modern parallel and distributed systems, the time for exchanging data is usually larger than that for computing elementary operations. Consequently, these communications slow down the execution of the application scheduled on such systems. Accounting for these communications is essential for attaining efficient hardware and software utilization. Therefore, we provide in this paper a new combined approach for scheduling parallel applications with large communication delays on an arbitrary number of processors. In this approach, a genetic algorithm is improved with the introduction of some extra knowledge about the scheduling problem. This knowledge is represented by a class of clustering algorithms introduced recently, namely, convex clusters which are based on structural properties of the parallel applications. The developed algorithm is assessed by simulations run on some families of synthetic task graphs and randomly generated applications. The comparison with related approaches emphasizes its interest.

1 Introduction

In modern parallel and distributed systems, the scheduling problem is more difficult not only by the new characteristics of these systems, but also by the need for data transfer among tasks, which in general are heavily communicated. Generally, the time for exchanging data is usually larger than that for computing elementary operations. Consequently, these communications slow down the execution of the application scheduled on the parallel processor system. Accounting for these communications is essential for attaining efficient hardware and software utilization. Given its importance, several heuristic methods have been developed for considering communications in the scheduling problem. The most widely used of the existing heuristics are *list scheduling*, *task clustering* and *genetic algorithms* [1]. Task clustering is an efficient way to reduce relatively the unbalance between communication delays and processing times. It starts by grouping heavily communicating tasks to the same labeled cluster considering unbounded of processors [2]. A post-clustering step is performed to obtain the final scheduling onto a bounded number of processors; however, if the post-clustering stage is performed without any attention it could degrade the overall system's performance when the number of available processors is less than the generated clusters [3].

In this paper, we combine task clustering with a meta-heuristic for efficient scheduling of applications with communication costs, especially for large communications, onto arbitrary number of processors. The main contribution is to develop an effective genetic algorithm, which is called *Genetic Acyclic Clustering Algorithm* (mGACA in short), for scheduling tasks through clustering directly. More precisely, it performs the clustering and post-clustering phases in one step. The major feature of the new algorithm is that it takes advantage of the effectiveness of task clustering for reducing communication delays combined with the ability of the genetic algorithms for exploring and exploiting information of the search space of the scheduling problem. The genetic algorithm makes the clustering based on structured properties of the parallel applications. Moreover, the main focus is on a recent class of structured clustering, called *convex*, which has interesting properties [4] and has been recently used in the context of uncertainties in the scheduling problem [5]. Hence, the final clustering generated by the genetic algorithm will also remain acyclic. The mGACA algorithm is assessed by simulations run on some families of traced task graphs representative of equation solver algorithms. Furthermore, the comparison with related approaches emphasizes its interest.

The rest of the paper is organized as follows. In Section 2, we state the problem, provide some related works and briefly discuss genetic algorithms. Section 3 introduces the new genetic algorithm. Experimental results are given in Section 4. Section 5 concludes the paper.

2 Background and Problem Statement

2.1 Notations and Definitions

Although many studies in scheduling have been focusing on heterogeneous systems, scheduling in homogeneous systems are still of concern because of its wide use and relative simplicity of modeling. In this paper, we consider a generic multiprocessor system composed of m identical processors linked by an interconnection network. While computing, a processor can communicate through one or several of its links and communications among tasks executed on the same processor are neglected. This computational model corresponds to the classical *delay model* [6].

As usually, the application is modeled by a *Directed Acyclic Graph* (DAG). It is represented by a precedence task graph $G = (V, E)$, where V is the set of n nodes corresponding to the tasks of the application that can be executed on the available processors. To every task t_j , there is an associated value p_j representing its processing time. $\mu(V) = \sum_{j \in V} p_j$ represents the time it would take to run all the tasks of the DAG on a single processor. E is the set of directed edges between the tasks that maintain a partial order among them. Let \prec be the partial order of the tasks in G , the partial order $t_i \prec t_j$ models precedence constraints. That is, if there is an edge $e_{i,j} \in E$ then task t_j cannot start its execution before task t_i completes. Hence, the results of task t_i must be available before task t_j starts its execution. The weight of any edge stands for the communication requirement

among the connected tasks. Thus, to every edge $e_{ij} \in E$ there is an associated value c_{ij} representing the time needed to transfer data from t_i to t_j . When we refer to large communication it means that $\max p_i < \min c_{ij}$.

Definition 1 (clustering). *A clustering $R = \{V_i, \prec_i\}_i$ of G is a mapping of the graph onto groups of tasks associated to a total linear order extension of the original partial order \prec .*

The considered scheduling objective is to minimize the makespan of a clustering R on the multiprocessor system when all the communication overheads are included. Unfortunately, finding a schedule of minimal makespan is in general a difficult problem, even if there are unbounded number of processors available [2]. An optimal schedule is a trade-off between high parallelism and low inter-processor communication. On the one hand, tasks should be distributed among the processors to balance the workload. On the other hand, the more nodes are distributed, the more inter-processor communications, which are expensive, are performed.

2.2 Related Works

There exists mainly two classes of task clustering algorithms: the algorithms based on the critical path analysis and those based on decomposition of the precedence task graph. The critical path based clustering algorithms try to shorten the longest execution path (considering both execution and communication costs) in a precedence task graph. The principle is to group tasks if they belong to the longest path and the relative parallel time is not increased [2, 7]. The clustering algorithms based on graph decomposition explicitly handle the trade-off between parallelism and communication overhead. They intent to divide the application into appropriate size and number of tasks to balance communication and parallelism so that the makespan is minimized. The principle of the graph decomposition clustering algorithms is to gather tasks into structured properties. McCreary and Gill in [8] developed a recursive canonical decomposition into *clans* (a group of tasks is a clan if all these tasks have the same predecessors and successors). This decomposition is unique and has also found applications in other fields such as a graph drawing.

Lepère and Trystram [4] provided a recursive decomposition based on the following principle: assigning tasks to locations into *convex clusters*. It means that for any path whose extremities are in the same cluster, all intermediate tasks are also in that cluster. The decomposition founded on convex clusters result in interesting properties. For example, considering arbitrary time execution and large communication delay the authors in [5] showed that the convex clustering are *3-Dominant*. It means that there exists a convex clustering whose execution time on an unbounded number of processors is less than three the time of an optimal clustering. Indeed, we expect better solutions on this specific class of structures. The authors in [5] also investigated convex clusters in the context of the scheduling problem with disturbances. In this context, the authors showed the ability of convex clusters to absorb the bad effects of disturbances on the

communications during the schedule without needing a stabilization process. The main assumption is that the resulting clustering founded on convex clusters does not contain any cycle (i.e., remains a DAG). Based on this assumption the authors in [5] claimed that any convex clustering algorithm is intrinsically stable since there are no cumulative effects of disturbances.

2.3 Genetic Algorithms and Clustering

Genetic Algorithms (GAs) are general-purpose, stochastic search methods where elements (called *individuals* or *chromosomes*) in a given set of solutions (called *population*) are randomly combined and modified (these combinations are called *crossover* and *mutation*) until some termination condition is achieved [9]. GAs use global search techniques to explore different regions of the search space simultaneously by keeping track of a set of potential solutions of diverse characteristics. GAs have been widely used for the scheduling problem in number of ways showing the potential of using this class of algorithms for scheduling [10]. Most of them can be classified as methods that use GA to evolve directly the actual assignment and order of tasks into processors, and methods that use GA in combination with other scheduling techniques. Zomaya and Chan [11] developed a genetic based clustering algorithm (henceforth called GA-cluster in short and will be used in the experimental section) which follows the principle of the Simple Genetic Algorithm [9]. GA-cluster takes as input the number of clusters to be generated and the schedule is obtained directly. A chromosome is represented as an array of integers of length equal to the number of tasks. Each entry in the array corresponds to the cluster for a task. The initial population is randomly created. Traditional genetic operators are used to generate new individuals. Selection uses a proportionate selection scheme.

3 Genetic Acyclic Clustering Algorithm: mGACA

3.1 The Provided Solution

The solution we provide is based on the decomposition of the task graph on convex clusters. That is, mGACA partitions the graph into convex sets of tasks with the property that the final clustering is a DAG (no cycles). The genetic operators could produce individuals with a number of clusters greater than the number of physical processors. Thus, a merging clustering algorithm is applied after the reproduction process until the number of clusters has been reduced to the actual number of processors. Both, genetic operators and merging clusters algorithm has been designed to produce only legal solutions not violating convexity constraints. Before presenting the new genetic algorithm, we introduce the notion of convex clusters following Lepère and Trystram in [4].

Definition 2 (Convex cluster). *A group of tasks $T \in V$ is convex if and only if all pairs of tasks $(t_x, t_z) \in T$, all the intermediate tasks t_y on any path between t_x and t_z belongs to T .*

For each clustering $R = \{V_i, \prec_i\}_i$, we can build a corresponding *cluster-graph* denoted by $G_R^{cluster}$ and defined as follows: the nodes of $G_R^{cluster}$ are the clusters of R . There exists an edge between two distinct cluster nodes V_i and V_j ($i \neq j$), if and only if there exist two tasks $t_x \in V_i$ and $t_y \in V_j$ such that $(t_x, t_y) \in E$. Moreover, the graph is weighted by $\max c_{xy}$ on each edge, and each cluster node V_i is weighted by $\nu(V_i) = \sum_{k \in V_i} p_k$.

Definition 3 (Acyclic clustering). A clustering R is acyclic, if and only if all the clusters are convex and $G_R^{cluster}$ remains a DAG.

3.2 Algorithm Design

The mGACA scheduling algorithm follows the principle of the Grouping Genetic Algorithms (GGA). GGA is a genetic algorithm heavily modified to suit the structure of grouping problems [12]. Those are the problems where the aim is to find a good partition of a set, or to group together the members of the set, but in most of these problems, not all possible groupings are allowed. GGA manipulates groups rather than individual objects, similarly mGACA manipulates clusters rather than tasks. The pseudo-code of the mGACA algorithm is given below:

String representation. An essential characteristic of GAs is the coding of the variables that describe the problem. In our GA-based approach, each chromosome is represented by a string of length n , where n is the number of tasks in the given DAG. Each gene of a given string in the chromosome is a tuple, such as (t_j, T_k) where the i -th location of the string includes the task t_j and its respective cluster T_k . We enriched this representation with a cluster part, encoding the clusters on one gene for one cluster basis. The length of the cluster part representation is variable and depends on the number of clusters (i.e., the physical processors) given in the input of mGACA, it means, the number of clusters is less than or equal to the number of processors. This representation is similar to that used in [12] to encode valid solutions for the grouping problems.

Initial population. mGACA uses a *Bottom-level Clustering Algorithm* (BCA) to create the initial population. BCA clusters the DAG according to the *Bottom level* (blevel) value of every task. The blevel of a task t_i is the length of the longest path from t_i to an exit task, including the processing time of t_i [13]. It is recursively defined by Eq.(1).

$$blevel(t_i) = p_i + \max_{t_j \in SUCC(t_i)} \{c_{ij} + blevel(t_j)\} \quad (1)$$

The BCA algorithm works as follows: the algorithm starts by randomly selecting the number of clusters nCl to be generated (between 1 and m) for any individual. After that, the clusters' grain size nt_Cl is calculated. Next, compute the *blevel* for each task and sort the tasks in decreasing order according to their *blevel* value (tie-breaking is done randomly). Afterward, assign the sorted tasks in a list. For each cluster T_i assign the nt_Cl tasks in the top of the list and remove them from that. This process is repeated until the number of clusters has been reached.

As mentioned above, any individual is legal if it does not contain any cycle respecting convexity and precedence constraints. The following proposition proves that BCA always generate legal individuals.

Proposition 1. *The BCA algorithm always generates feasible acyclic clustering.*

Proof. The proof is as follows. Assume that T_1, \dots, T_l are clusters generated by BCA and tasks $t_i \in T_i, t_j \in T_j$, for $i < j$. Since BCA sorts and clusters tasks in sequence, it is obvious that $blevel(t_i) > blevel(t_j)$. From the definition of *blevel*, t_i cannot be a successor of t_j . Thus, tasks in T_i will not depend on any task in T_j . Therefore, the clusters are convex and the resulting clustering is acyclic. \square

Fitness Function. We used the fitness function of the GA-cluster algorithm in [11]. To evaluate the fitness of each solution, first the schedule length of the particular solution is determined, then this schedule length is mapped to an initial fitness value (f'_{S_i}) using Eq.(2):

$$f'_{S_i} = \mu(V) - C_{S_i} \quad (2)$$

Where C_{S_i} represents the schedule length of the individual S_i . However, in some cases f'_{S_i} can still be negative since some of the initial clusterings can result in schedules whose makespan is worse than running the tasks on a single processor. So a linear scaling is employed to obtain the final fitness values (Eq.(3)).

$$f_{S_i} = \frac{f'_{S_i} - \text{MIN}[f'_{S_i}]}{\text{MAX}[f'_{S_i}] - \text{MIN}[f'_{S_i}]} \quad (3)$$

Eq.(3) is then applied to all initial fitness values to map them into the final fitness values, and evaluate the individuals in the population. To obtain the schedule length for a particular clustering, each task is scheduled in decreasing order priority. That is, to schedule a task into the first slot available after the specified earliest start time on the assigned processor. The earliest start time $est(t_i)$ for task t_i is calculated using Eq.(4):

$$est(t_i) = \text{MAX}[\sigma(t_i) + p_i + c_{ij}, (i, j) \in E] \quad (4)$$

Where: $\sigma(t_i)$ is the starting execution time of task t_i . Let us recall that $c_{ij} = 0$ if tasks t_i and t_j are placed on the same processor. Once all tasks have been scheduled, the algorithm uses Eq.(5) to determine the schedule length of the scheduled DAG.

$$C_{S_j} \equiv \text{MAX}[\sigma(t_j) + p_j] \quad (5)$$

Stopping Condition. There are three ways mGACA will possibly terminate. Firstly, if it has reached the maximum number of iterations. Secondly, when the best fitness in a population has not changed for a specified number of generations. Finally, when the difference between the average and the best fitness remains constant for some given number of generations.

Selection. mGACA uses the proportionate selection scheme and replaces the entire previous generation. In order to implement proportional selection, mGACA uses the *roulette wheel* principle of selection. In the roulette wheel selection, the probability for selecting an individual S_i is directly proportional to its fitness.

Crossover. Crossover takes two individuals as input and produces new individuals that have some portions of both parent's genetic material. Hence, the offspring keeps some of the characteristics of the parents. We adapted the crossover algorithm given in [12]. Let S_1 and S_2 be individuals which should generate offspring S'_1 and S'_2 . The algorithm starts by selecting randomly two crossing sites and delimiting the *crossing section*, in each of the two parents. Then, inject the contents of the crossing section of S_2 (that is, the second parent) in the first crossing site of S_1 (the first parent). Recall that the crossover works with the cluster part of the chromosome, so this means injecting some of the *clusters* from S_2 into S_1 . After that, eliminate all *tasks* now occurring twice from the clusters they were belonging to in the first parent. Consequently, some of the "old" clusters coming from the first parent are altered: they do not contain all the same tasks anymore since some of those tasks had to be eliminated. If necessary, *adapt* the resulting clusters, according to the convex cluster constraint. At this stage, a local problem-dependent heuristic has been used. Finally, apply the same procedure to the two parents with their roles permuted in order to generate the second offspring (i.e., the second child).

The proposed crossover algorithm always generate feasible convex clusters since another local heuristic is used to repair the altered clusters, "*if necessary*". We have developed a simple local heuristic as follows: we remove the task of the altered cluster violating the convex constraint and allocate it and the set of its predecessors into a new cluster. We only select the predecessors belonging to the altered cluster.

Merging. Since crossover operator can generate chromosomes with a number of clusters greater than the available physical processors, we developed an algorithm to merge existing clusters in a pairwise fashion until the number of clusters is reduced to m . The merging clusters algorithm (MCA) works over the cluster-graph $G_R^{cluster}$. Recall that the clusters represent the task nodes in $G_R^{cluster}$. This procedure initially sorts the elementary clusters by $\nu(V_i)$, to allow the smallest cluster node to be considered for merging at each step. For each cluster calculates its blevel. After that, chooses the smallest cluster. If the chosen cluster will be merged with one of its immediate predecessors, then select the predecessor with the smallest blevel (ties are broken by selecting the immediate predecessor which communicates the most) and merge them in one cluster. In other case, if the chosen task will be merged with one of its immediate successors, then select the immediate successor with the greatest blevel value and merge them in one cluster. Repeat this process until the number of cluster nodes is less than or equal to the number of processors m . Let us remark that, if the chosen task is an entry task

(i.e., a task without predecessors) then the algorithm merges it with any of its immediate successors. In other case, if it is an exit task (i.e., a task without successors) then MCA chooses a task among its immediate predecessors and merges them.

Proposition 2. *MCA always merge $G_R^{cluster}$ in feasible acyclic clustering.*

Proof. Let us proof Proposition 2 with an example. Let us consider the acyclic graph $G_R^{cluster}$. Supposse that we have three cluster nodes of $G_R^{cluster}$ (we name them T_x , T_y and T_z , respectively). Let us consider the partial orders $T_x \prec T_y \prec T_z$ and $T_x \prec T_z$ given in the $G_R^{cluster}$ graph. Now, consider that T_z is selected to be merged with another cluster. Thereafter, $T_x, T_y \in PREC(T_z)$ (the set of immediate predecessors). It is clear that $blevel(T_y) < blevel(T_x)$. As MCA merges only two clusters at the same time and selects always the $\min(blevel_{\forall i \in PREC(T_z)}[i])$, thus MCA will select T_y to merge with T_z and the resulting clustering still acyclic clustering. As it holds for all $T_y \in PREC(T_z)$ with the $\min(blevel_{\forall T_y \in PREC(T_z)}[T_y])$ this completes the proof. \square

Mutation. The mutation operator works by changing a task's cluster to a new cluster. First, an individual is randomly chosen. Next, the mutation operator is applied to the selected chromosome with a certain probability p_m . Then, the mutation operator selects a cluster T_i randomly (from the cluster part of the individual). After that, the mutation operator selects a task t_i from the top or the bottom of the selected cluster T_i , creates a new cluster and puts the task t_i in the new cluster. We recall that the tasks in the clusters are sorted by topological order. Thus, if the mutation operator selects any task from the top or the bottom of any cluster and puts it in a new cluster does not violate the convex cluster constraint and the resulting clustering is acyclic. Finally, after mutation, mGACA verifies if the number of clusters is greater than the number of processors m , then apply the merging algorithm MCA.

4 Experiments

In this section, we present the performance comparison of mGACA and related algorithms by extensive experiments. The related algorithms are the well-known standard algorithm *ETF* [14] and *GA-cluster* described in Section 2.3. Experimentations have been performed on a PC with a 1 Ghz dual-core. The following parameter values were adopted by mGACA and GA-cluster for these experiments: population size equal to 300, crossover probability of 0.75, mutation probability of 0.001, a generation limit of 300, and a percentage of convergence of 0.75. We conducted experiments on two classes of instances: synthetic task graphs, and random graphs. We have used the normalized schedule length (NSL) as a metric for comparison. It is defined as the ratio of the schedule length computed by the algorithm and the sum of the processing times of the tasks on a critical path. Average NSL (ANSL) values are used in our experiments.

4.1 Results on Synthetic Task Graphs

For the first comparison, we present the schedule lengths produced by each algorithm for a set of synthetic task graphs representing basic numeric kernels [15]. Synthetic task graphs can be characterized by the size of the input data matrix because the number of nodes and edges in the task graph depends on the size of this matrix. For example, the number of nodes for LU decomposition graph is equal to $(N^2 + N - 2)/2$ where N is the size of the matrix. We conducted experiments on three different families of synthetic graphs: namely LU decomposition, Cholesky factorization, and Jacobi transformation. For each synthetic graph, we have varied the communication to computation cost ratio (CCR). It is defined as the average of weights of edges divided by the average of tasks' computation times. Figure 1 gives the ANSL values of the algorithms for the Cholesky graphs at various matrix sizes when the number of processors is equal to 8 and 32. For each matrix size, there are three different CCR ratios: 5, 10 and 15. We observe that the performance of the mGACA algorithm outperforms on average the related approaches. For example, the ANSL value of mGACA on 8 processors, is shorter than that of the ETF and GA-cluster algorithms by 12.44% and 50% respectively. Moreover, the ANSL value produced by mGACA on 32 processors, is shorter than that of the ETF and GA-cluster by 15.23% and 61% respectively.

The next experiment is with respect to various CCR ratios. We simulated the algorithms on the LU graph with matrix size equal to 19 (189 tasks) and six CCR values 5, 8, 10, 12, 15, 18. Figure 2 shows the performance of mGACA with the result of ETF and GA-cluster by computing a percentage of improvement of the ANSL:

$$gain = \frac{ANSL_{algorithm} - ANSL_{mGACA}}{ANSL_{algorithm}} * 100 \quad (6)$$

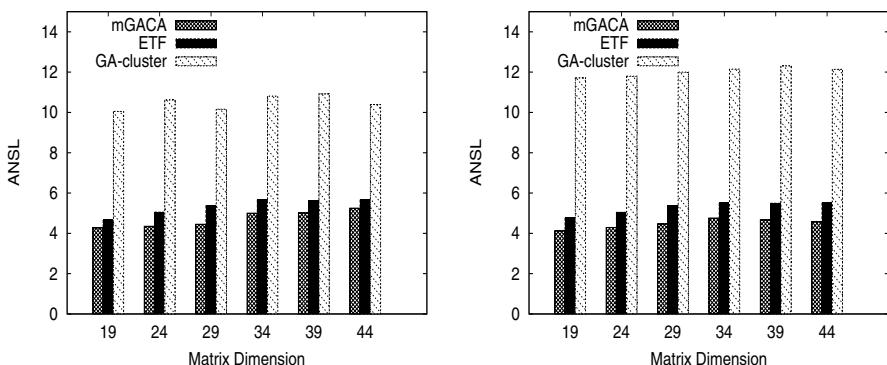


Fig. 1. ANSL for Cholesky factorization graphs with respect to the matrix size on 8 (left) and 32 (right) processors

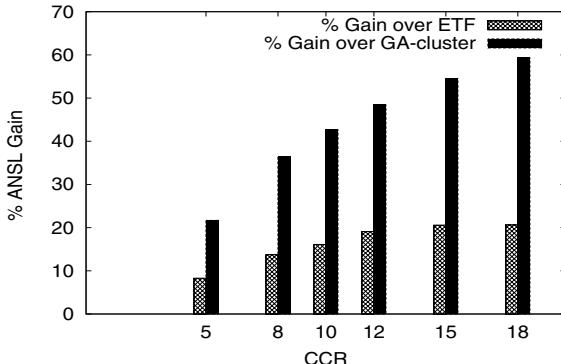


Fig. 2. Percentage improvement in ANSL of mGACA over ETF and GA-cluster

From Figure 2, we notice that mGACA improves the performance of ETF and GA-cluster while increases the CCR. Let us remark that for this experiment mGACA never computed schedules that delayed the makespan (i.e., schedules greater than the sequential time) when scheduling on the LU graphs which is not the case for ETF and GA-cluster.

4.2 Results on Random Graphs

This subsection presents the results obtained on random graphs. We have generated two sets of random graphs: Winkler graphs and Layered random graphs. The Winkler graphs [16] are random graphs representative of multidimensional orders. The three algorithms have been tested on random 2-dimensional orders. To generate 2-dimensional order with n elements, n points are chosen randomly in the $[0; 1] \times [0; 1]$ square. Each point becomes a node and there is an arc between two points a and b if b is greater than a according both dimensions. To generate the second set of random graphs we have implemented a layered random graph generator such as the one proposed in [7]. The algorithm takes three parameters N , L and p that can be described as follow: N nodes are distributed on L layers and for each couple of successive layers $(l, l + 1)$ and each couple $(a + b) \in l \times (l + 1)$, there is an arc (a, b) with probability p . No other arcs are present (especially between non successive layers). For generating the two sets of random graphs, we varied two parameters: size and CCR. The size of the graph was varied from 100 to 300 nodes with increment of 50. Five different values of CCR were selected: 2, 5, 10, 15, and 20. There were 30 graphs generated for each combination of those parameters. The processing time was randomly selected from the range $(1, 20)$ with the uniform distribution. The algorithms were scheduling those graphs on three different number of processors (fully connected 8, 16 and 32 processors). In total, there were 2250 different experiment settings. We compared the result of the mGACA with the result of ETF and GA-cluster by computing the percentage of improvement of the ANSL.

Table 1. Percentage gain on the normalized schedule length produced by mGACA over the ETF and GA-cluster

Factor		% gain over ETF			% gain over GA-cluster		
		Best	Avg	Worst	Best	Avg	Worst
CCR	2	-17.25	-19.25	-22.25	-12.02	-11.33	-10.24
	5	-1.32	2.66	6.43	21.585	22.57	23.95
	10	5.23	18.10	25.52	50.02	50.73	51.58
	15	9.71	19.47	28.12	61.34	61.8	62.39
	20	11.77	23.67	32.76	67.90	68.26	68.78
graph size	100	-1.88	7.39	14.60	52.33	54.28	56.02
	150	0.0	10.25	17.11	56.67	57.11	57.80
	200	1.00	11.68	20.37	54.06	54.43	55.01
	250	3.74	15.24	24.09	50.45	50.83	51.33
	300	7.33	14.18	20.64	46.08	46.50	47.13
system	8	3.31	12.03	18.94	48.82	49.88	50.98
	16	3.64	12.43	20.04	52.13	52.73	53.48
	32	5.00	12.57	20.99	53.72	54.08	54.66

We ran each algorithm 20 times on each graph. We compared the best, average and the worst *ANSL*. As the number of experiments is considerable, Table II presents only aggregated results. Negative values mean that mGACA gets largest schedules than that computed by the related algorithms. We observe that any increases of the different experiment settings (i.e., CCR, size) the performance of mGACA increases regarding ETF and GA-cluster. We can see that on average mGACA outperforms the related approaches. The most significant improvement is given while increasing communication delays. The mGACA algorithm performs better on graphs with large communications. On the contrary, it is the most important factor that affects the performance of the related approaches. On the one hand, the mGACA algorithm takes advantage of clustering by gathering tasks into appropriate size to balance communication and parallelism while minimizing the makespan. On the other hand, it is well-known that the performance of ETF is $2 - \frac{1}{m} + \rho$ factor of the optimal [14], where ρ is a factor of the communication delays. If communication delays increase the makespan of the ETF algorithm increases as well. With respect to the GA-cluster algorithm, the knowledge-augmented significantly helps mGACA to locate better solutions than GA-cluster.

Regarding the running time, we noticed as expected that mGACA is slower than ETF because the nature of genetic algorithms. However, the running time to compute a solution is admissible in all the experiments, for example, for the Cholesky factorization graph with matrix size of 44 (1034 tasks), mGACA spent 35 seconds on average to compute a solution. On the other hand, the running time of mGACA is smaller than GA-cluster because the knowledge-augmented significantly helps mGACA to explore the search space and locate better solutions spending less time than GA-cluster.

5 Conclusions and Perspectives

We have presented a new genetic algorithm for scheduling tasks through clustering based on the graph decomposition. The new algorithm was improved with the introduction of some knowledge about the scheduling problem. This knowledge is represented by the convex clusters which are based on structural properties of the parallel applications. Extensive experiments have been run for comparing the developed approach to popular related scheduling algorithms and emphasize the interest of this algorithm and convex clusters. The new genetic algorithm seems well suited for new parallel systems like clusters of workstations where the inter-processor communications are much larger than the intra-processor communication costs. One important prospect is to investigate the behavior of the developed algorithm in the context of the scheduling with disturbances on the communication delays.

References

1. Sinnen, O.: *Task Scheduling for Parallel Systems*. Wiley-Interscience, NJ (2007)
2. Sarkar, V.: *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, Cambridge (1989)
3. Kianzad, V., Bhattacharyya, S.S.: Efficient techniques for clustering and scheduling onto embedded multiprocessors. *IEEE TPDS* 17(7), 667–680 (2006)
4. Lepère, R., Trystram, D.: A new clustering algorithm for large communication delays. In: Proc. 11th IPDPS 2002, Fort Lauderdale, Florida, April 2002, pp. 68–73 (2002)
5. Mahjoub, A., Pecero, J.E., Trystram, D.: Scheduling with uncertainties on new computing platforms. *Journal Comput. Optim. Appl.* (to appear)
6. Rayward-Smith, V.J.: Uet scheduling with unit interprocessor communication delays. *Discrete Applied Mathematics* 18(1), 55–71 (1987)
7. Yang, T., Gerasoulis, A.: Dsc: Scheduling parallel tasks on an unbounded number of processors. *IEEE TPDS* 5(9), 951–967 (1994)
8. McCreary, C., Gill, H.: Automatic determination of grain size for efficient parallel processing. *Comm. of ACM* 32(9), 1073–1078 (1989)
9. Goldberg, D.E.: *Genetic algorithms in search, optimization, machine learning*. Addison-Wesley, Boston (1989)
10. Zomaya, A.Y., Ercal, F., Olariou, S. (eds.): *Solutions to parallel and distributed computing problems: Lessons from biological sciences*. Wiley, NY (2001)
11. Zomaya, A.Y., Chan, G.: Efficient clustering for parallel tasks execution in distributed systems. In: Proc. NIDISC 2004, IPDPS (April 2004)
12. Falkenauer, E.: *Genetic algorithms and grouping problems*. John Wiley and Sons Ltd., England (1999)
13. Kwok, Y.K., Ahmad, I.: Benchmarking and comparison of the task graph scheduling algorithms. *JPDC* 59(3), 381–422 (1999)
14. Hwang, J.J., Chow, Y.C., Angers, F.D., Lee, C.Y.: Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal on Computing* 18(2), 244–257 (1989)
15. Kitajima, J.P., Plateau, B., Bouvry, P., Trystram, D.: Andes: Evaluating mapping strategies with synthetic programs. *J. of Syst. Arch.* 42(5), 351–365 (1996)
16. Winkler, P.: Random orders. *Order* 1, 317–331 (1985)

Comparison of Access Policies for Replica Placement in Tree Networks

Anne Benoit

LIP Laboratory, ENS Lyon, 46 allée d'Italie, 69364 Lyon Cedex 07, France
Université de Lyon, UMR 5668 ENS Lyon-CNRS-INRIA-UCBL

Anne.Benoit@ens-lyon.fr

Abstract. In this paper, we discuss and compare several policies to place replicas in tree networks subject to server capacity. The server capacities and client requests are known beforehand, while the number and location of the servers are to be determined. The standard approach in the literature is to enforce that all requests of a client be served by a single server in the tree (*Single*). One major contribution of this paper is to assess the impact of a new policy in which requests of a given client can be processed by multiple servers (*Multiple*), thus distributing the processing of requests over the platform. We characterize problem instances for which *Multiple* cannot be more than two times better than the optimal *Single* solution, if this latter exists. For such instances, we provide a procedure which builds a *Single* solution with a guarantee on its cost. This is a very interesting result for applications which do not accept multiple servers for a given client, since it might be more difficult to implement such a complex strategy.

Keywords: replica placement, access policies, heterogeneous platforms, hierarchical platforms, video-on-demand applications.

1 Introduction

In this paper, we discuss and compare several policies to place replicas in tree networks subject to server capacity. The server capacities and client requests are known beforehand, while the number and location of the servers are to be determined. A *client* is a leaf node of the tree, and its requests can be served by one or several *internal nodes*, which have a fixed capacity. Initially, there are no replica; when an internal node of the tree is equipped with a replica, it can process a number of requests, up to its capacity limit. Nodes equipped with a replica, also called *servers*, can only serve clients located in their subtree (so that the root, if equipped with a replica, can serve any client); this restriction is usually adopted to enforce the hierarchical nature of the target application platforms, where a node has knowledge only of its parent and children.

We also point out that the distribution tree (clients and nodes) is fixed in our approach¹. This key assumption is quite natural for a broad spectrum of

¹ Note that we further assume that client requests are known beforehand and do not change over time. We keep the study of a dynamic setting for future work.

applications, such as electronic, internet service provider (ISP), or video-on-demand (VOD) service delivery [123]: a distribution tree is deployed to provide a hierarchical and distributed access to replicas. On the contrary, in other more decentralized applications (e.g., allocating Web mirrors in distributed networks), a two-step approach is used: first determine a “good” distribution tree in an arbitrary interconnection graph, and then determine a “good” placement of replicas among the tree nodes. Both steps are interdependent, and the problem is much more complex, due to the combinatorial solution space (the number of candidate distribution trees may well be exponential).

The rule of the game is to assign replicas to nodes so that some optimization function is minimized. Typically, this optimization function is the total utilization cost of the servers. If all nodes are identical (homogeneous version), this reduces to minimizing the number of replicas. If the nodes are heterogeneous, it is natural to assign a cost proportional to their capacity (so that one replica on a node capable of handling 200 requests is equivalent to two replicas on nodes of capacity 100 each).

The standard approach in the literature (see [4] for instance) is to enforce that all requests of a client be served by a single server in the tree; this policy is called *Single*. Following the hierarchical structure of the platform, the server must be on the path from the client to the root of the tree. We introduced in [5] a new policy, *Multiple*, in which the requests of a given client can be processed by multiple servers on the path, thus distributing the processing of requests over the platform. All problems were shown to be NP-complete, except the *Multiple*/homogeneous combination, in which the optimal solution can be found in polynomial time, using a multi-pass greedy algorithm (see [5]).

One major contribution of this paper is to assess the impact of the new *Multiple* policy on the total replication cost, and the impact of server heterogeneity, both from a theoretical and a practical perspective. Thus we demonstrate the usefulness of the new policy, even in the case of identical servers. The first result is that *Multiple* allows us to find solutions when the classical single policy does not. More generally, a single server policy will never have a solution if one client sends more requests than the largest server capacity. Then we build an instance of the problem where both access policies have a solution, but the solution of *Multiple* is arbitrarily better than the solution of *Single*. This is quite evident for heterogeneous platforms [5], but it is a new result for the homogeneous case.

If we focus on homogeneous platforms, there are many problem instances for which *Multiple* cannot be more than two times better than the optimal *Single* solution, if this latter exists. In our work, we thoroughly characterize such cases and we provide a procedure which builds a *Single* allocation with a guarantee on the cost. The idea consists in having a single server allocation in which each server is processing a number of requests being at least equal to half of its processing capacity. The servers may be fully used in the *Multiple* allocation, thus the solution will be up to two times better, but not arbitrarily better. This is a very interesting result for applications which do not accept multiple servers for a given client.

The rest of the paper is organized as follows. Section 2 is devoted to a detailed presentation of the target optimization problems. In Section 3 we compare the different access policies. Next in Section 4 we proceed to the complexity results, both in the homogeneous and heterogeneous cases. Section 5 introduces the procedure to build efficient *Single* solutions. Section 6 is devoted to an overview of related work. Finally, we state some concluding remarks in Section 7.

2 Framework

This section is devoted to a precise statement of the replica placement optimization problem. We start with some definitions and notations. Next we outline the simplest instance of the problem. Then we describe several types of constraints that can be added to the formulation.

2.1 Definitions and Notations

We consider a distribution tree whose nodes are partitioned into a set of clients \mathcal{C} and a set of nodes \mathcal{N} . The clients are leaf nodes of the tree, while \mathcal{N} is the set of internal nodes. It would be easy to allow *client-server* nodes which play both the role of a client and of an internal node (possibly a server), by dividing such a node into two distinct nodes in the tree. Each client $i \in \mathcal{C}$ (leaf of the tree) is sending r_i requests per time unit to a database object. A node $j \in \mathcal{N}$ (internal node of the tree) may or may not have been provided with a replica of this database. Nodes equipped with a replica (i.e., servers) can process requests from clients in their subtree. In other words, there is a unique path from a client i to the root of the tree, and each node in this path is eligible to process some or all the requests issued by i when provided with a replica. Node $j \in \mathcal{N}$ has a processing capacity W_j , which is the total number of requests that it can process per time unit when it has a replica. A cost, sc_j , is also associated to each internal node, which represents the price to pay to place a replica at this node. It is quite natural to assume that sc_j is proportional to W_j : the more powerful a server, the more costly.

Let r be the root of the tree. If $j \in \mathcal{N}$, then $\text{children}(j) \subseteq \mathcal{N} \cup \mathcal{C}$ is the set of children of node j . If $k \neq r$ is any node in the tree (leaf or internal), $\text{parent}(k) \in \mathcal{N}$ is its parent in the tree. Let $\text{ancestors}(k) \subseteq \mathcal{N}$ denote the set of ancestors of node k , i.e., the nodes in the unique path that leads from k up to the root r (k excluded). Finally, $\text{subtree}(k) \subseteq \mathcal{N} \cup \mathcal{C}$ is the subtree rooted in k , including k .

2.2 Problem Instances

There are two scenarios for the number of servers assigned to each client:

Single server – Each client i is assigned a single server that is responsible for processing all its requests.

Multiple servers – A client i may be assigned several servers in a set $\text{servers}(i)$.

To the best of our knowledge, the single server policy (*Single*) has been enforced in all previous approaches. One objective of this paper is to assess the impact of this restriction on the performance of data replication algorithms. The single server policy may prove a useful simplification, but may come at the price of a non-optimal resource usage (see Section 3).

For each client $i \in \mathcal{C}$, let $\text{servers}(i) \subseteq \text{ancestors}(i)$ be the set of servers responsible for processing at least one of its requests. In a single server access policy, this set is reduced to only one server: $\forall i \in \mathcal{C} \mid \text{servers}(i) \mid = 1$. Let R be the set of servers, i.e., nodes equipped with a replica: $R = \{s \in \mathcal{N} \mid \exists i \in \mathcal{C}, s \in \text{servers}(i)\}$. Also, we let $r_{i,s}$ be the number of requests from client i processed by server s . All requests must be processed, thus $\sum_{s \in \text{servers}(i)} r_{i,s} = r_i$. In the single server case, this means that a unique server is handling all r_i requests. The problem is constrained by the server capacities: no server capacity can be exceeded, thus $\forall s \in R, \sum_{i \in \mathcal{C} \mid s \in \text{servers}(i)} r_{i,s} \leq W_s$. Finally, the objective function is defined as $\min \sum_{s \in R} \text{sc}_s$.

In addition to the two access policies *Single* or *Multiple*, we consider different platform types, with different or identical servers:

Different servers – As already pointed out, it is frequently assumed that the cost of a server is proportional to its capacity. The problem thus reduces to finding a valid solution of minimal cost, where “valid” means that no server capacity is exceeded. In this case, we can scale the storage costs such that $\text{sc}_s = W_s$ since we minimize a sum of storage costs. We call this problem **REPLICA COST**.

Identical servers – We can further simplify the previous problem in the homogeneous case: with identical node capacities ($\forall s \in \mathcal{N}, W_s = W$), the **REPLICA COST** problem amounts to minimize the number of servers needed to solve the problem. In this case, the storage cost sc_j is set to 1 for each node. We call this problem **REPLICA COUNTING**.

2.3 Note about the *Closest* Policy

In the literature (see [4] for instance), the *Single* strategy is further constrained to the *Closest* policy: the server of client i is constrained to be the first server found on the path that goes from i upwards to the root of the tree. In particular, consider a client i and its server s . Then any other client i' residing in $\text{subtree}(s)$ will be assigned a server in that subtree. This forbids requests from i' to “traverse” s and be served higher (closer to the root in the tree).

We relax this constraint in the *Single* policy. Note that a solution to *Closest* always is a solution to *Single*, thus *Single* is always better than *Closest* in terms of the objective function. Similarly, the *Multiple* policy is always better than *Single*, because it is not constrained by the single server restriction.

3 Access Policies Comparison

In this section, we compare the general *Single* policy with the *Multiple* one for the **REPLICA COUNTING** problem. Section 3.1 illustrates the impact of policies

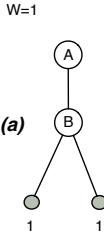


Fig. 1. Solution existence

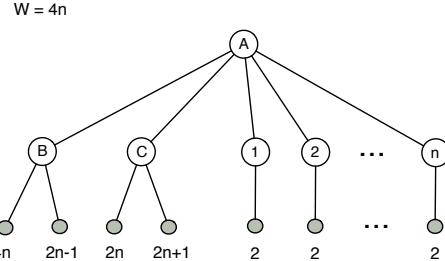
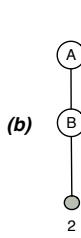


Fig. 2. Solution cost

on the existence of a solution. Then Section 3.2 shows that *Multiple* can be arbitrarily better than *Single*. Finally, Section 3.3 presents lower bounds and show some cases in which the optimal solution is arbitrarily higher than this bound. This comparison is done on the REPLICA COUNTING problem, thus the results can be generalized to the REPLICA COST problem.

3.1 Impact of the Access Policy on the Existence of a Solution

We consider here a very simple instance of the REPLICA COUNTING problem. In this example there are two nodes, *B* being the unique child of *A*, the tree root (see Fig. 1). Each node can process $W = 1$ request.

If *B* has two clients, each making one request, the solution consists in placing a replica on each node, and both clients can be served (Fig. 1(a)). This works for the *Single* policy (and thus for the *Multiple* one). However, if *B* has only one client making two requests, only *Multiple* has a solution since we need to process one request on *A* and the other on *B*, thus requesting multiple servers (Fig. 1(b)).

This example demonstrates the usefulness of the new policy: the *Multiple* policy allows to find solutions when the classical *Single* policy does not. More generally, a single server policy will never have a solution if one client sends more requests than the largest server capacity.

3.2 Impact of the Access Policy on the Cost of a Solution

In this section we build an instance of the REPLICA COUNTING problem where both access policies have a solution, but the solution of *Multiple* is arbitrarily better than the solution of *Single*.

Consider the instance of REPLICA COUNTING represented in Fig. 2, with $3+n$ nodes of capacity $W = 4n$. The root *A* has $n+2$ children nodes *B*, *C* and $1, \dots, n$ ($n \geq 1$). Node *B* has two clients, one with $4n$ requests and the other with $2n-1$ requests. Node *C* has two clients, one with $2n$ requests and the other with $2n+1$ requests. Each node numbered *i* has a unique child, a client with 2 requests.

The *Multiple* policy assigns 3 replicas to *A*, *B* and *C*. *B* handles the $4n$ requests of its first client, while the other client is served by *A*. *C* handles $2n$ requests

from both of its clients, and the 1 remaining request is processed by A . Server A therefore processes $(2n-1)+1 = 2n$ requests coming up from B and C . Requests coming from the n remaining nodes sum up to $2n$, thus A is able to process all of them.

For the *Single* policy, we need to assign replicas everywhere. Indeed, with this policy, C cannot handle more than $2n+1$ requests since it is unable to process requests from both of its children, and thus A has $(2n-1)+2n$ requests coming from B and C . It cannot handle any of the $2n$ remaining requests, and thus each remaining node must process requests coming from its own client. This leads to a total of $n+3$ replicas.

The performance factor is thus $\frac{n+3}{3}$, which can be arbitrarily big when n becomes large.

3.3 Lower Bound

Obviously, the cost of an optimal solution of the REPLICATING problem (for any policy) cannot be lower than the lower bound $\left\lceil \frac{\sum_{i \in C} r_i}{W} \right\rceil$, where W is the server capacity. Indeed, this corresponds to a solution where the total request load is shared as evenly as possible among the servers.

The example of Fig. 3 shows that the solution can be arbitrarily higher than this lower bound, even for the *Multiple* policy, since many servers are required to be placed and each of them is only handling a small portion of work.

Consider Fig. 3, with $n+2$ nodes of capacity $W = n > 0$. The root of the tree A has $n+1$ children: B and $1, \dots, n$. Node B has two clients, each sending n requests. Each other node has a unique child, a client with only one request. The lower bound is $\left\lceil \frac{\sum_{i \in C} r_i}{W} \right\rceil = \frac{3n}{n} = 3$. However, both policies will assign replicas to A and B to cover both clients of B , and will then need n extra replicas, one per node $1, \dots, n$. The total cost is thus $n+2$ replicas, arbitrarily higher than the lower bound.

Previous examples give an insight of the combinatorial nature of the replica placement optimization problems, even in its simplest variant with identical servers, REPLICATING. The following section corroborates this insight: most problems are shown NP-hard, even though some variants have polynomial complexity.

4 Complexity Results

The decision problems associated with the previous optimization problems are easy to formulate: given a bound on the number of servers (homogeneous version)

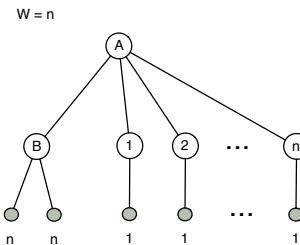


Fig. 3. The lower bound cannot be approximated for REPLICATING

Table 1. Complexity results

	<i>Single</i>	<i>Multiple</i>
REPLICA COUNTING	NP-complete	polynomial
REPLICA COST	NP-complete	NP-complete

or on the total storage cost (heterogeneous version), is there a valid solution that meets the bound?

Table II captures the complexity results, see [5] for the proofs. The NP-completeness of the *Single*/REPLICA COUNTING case comes as a surprise, since all previously known instances with the extra *Closest* constraint were shown to be polynomial, using dynamic programming algorithms. With different servers (REPLICA COST), the problem is combinatorial due to resource heterogeneity. The only polynomial case is the *Multiple*/REPLICA COUNTING combination, in which the optimal solution can be found using a multi-pass greedy algorithm described in [5].

5 From *Multiple* to *Single*

In this section, we give a procedure to build a *Single* allocation for the REPLICA COUNTING problem, with a guarantee on the cost. Indeed, while an optimal solution for the *Multiple* problem can be found in polynomial time, the *Single* problem is NP-complete. Still, some applications may require a *Single* policy since the implementation may be less complex for such a policy. Our goal is thus to build a good *Single* allocation, compared to a lower bound given by the *Multiple* solution.

Recall that in the worst case, the optimal *Single* can be arbitrarily worse than the optimal *Multiple* (see Section 3.2). Thus we aim at characterizing problem instances for which a good *Single* solution can be derived.

Problem formulation: Let $(\mathcal{C}, \mathcal{N})$ be a problem instance in which $r_i \leq W$ for all $i \in \mathcal{C}$ (otherwise, there is no solution to the *Single* problem). We are given an optimal *Multiple* solution for this problem, of cost M (i.e., M is the number of servers in this solution). We aim at finding a *Single* solution with a cost $S \leq 2M$, and at characterizing cases in which this is possible.

5.1 Linear Trees

First let us consider a linear tree consisting in $|\mathcal{N}| = n$ nodes and rooted in node 1 such that $1 \rightarrow 2 \rightarrow \dots \rightarrow n$, and a set \mathcal{C}_j of clients attached to node j of the tree, for $1 \leq j \leq n$ (note that $\mathcal{C} = \cup_{1 \leq j \leq n} \mathcal{C}_j$). Furthermore, we assume that $jW \geq 2 \sum_{i \in \cup_{1 \leq k \leq j} \mathcal{C}_k} r_i$, meaning that, at each level of the tree, there are twice more nodes than the minimum number of servers requested to handle all requests from the root down to this level. On the whole tree, we have a condition on the total number of nodes, $n \geq \frac{2}{W} \sum_{i \in \mathcal{C}} r_i$.

We build a solution to *Single* by assigning requests greedily, starting from the clients at the top of the tree, since they have less choice of nodes where to be processed. At each step, we try to give requests to servers that are already processing some requests, and we try to allocate requests starting from the bottom of the tree. If it fails, a new server is created on the first free node. The **linear-tree** procedure returns the *Single* solution in which $s_{i,j} = r_i$ if node j is processing requests of client i , and s_j is the total number of requests processed by node j , $s_j = \sum_{i \in \mathcal{C}} s_{i,j}$.

```

procedure linear-tree ( $\mathcal{C}, \mathcal{N}$ )
 $\forall i \in \mathcal{C}, \forall j \in \mathcal{N}, s_{i,j} = 0; \forall j \in \mathcal{N}, s_j = 0; \quad // \text{Initialisation.}$ 
for  $j = 1..n$  do
  for  $i \in \mathcal{C}_j$  do
    // Loop 1: try to add requests to an existing server.
    for  $j' = j..1$  do
      if  $\sum_{k \in \mathcal{N}} s_{i,k} = 0$  and  $s_{j'} \neq 0$  then
        if  $r_i + s_{j'} \leq W$  then  $s_{i,j'} = r_i; s_{j'} = s_{j'} + r_i;$ 
      end
    end
    // Loop 2: If Loop 1 did not succeed, create a new server.
    if  $\sum_{k \in \mathcal{N}} s_{i,k} = 0$  then
      for  $j' = j..1$  do
        if  $s_{j'} = 0$  then  $s_{i,j'} = r_i; s_{j'} = r_i; \text{break};$ 
      end
    end
  end
end
return  $\{s_{i,j} \mid i \in \mathcal{C}, 1 \leq j \leq n\}$ 

```

For each client, all its requests are processed by a single server, either added to an existing server (Loop 1), or to a new server (Loop 2). The procedure is correct if and only if Loop 2 always succeed to find a new server, and this loop never fails because of the assumption on the number of nodes. Indeed, let us count the number of servers allocated by the procedure, S_j , at each step of the outer loop on j , and let us prove that $S_j \leq j$. This means that for all j there are enough nodes available so that we are able to perform the allocation.

Proof: Each couple of servers (k, k') is such that $s_k + s_{k'} > W$, otherwise the greedy allocation would have assigned all requests to one of these servers. Thus, all servers but eventually the last one are handling at least $W/2$ requests. If $S_j = 1$, then $S_j \leq j$ since $j \geq 1$, and we are done. If $S_j \geq 2$, we associate the last server, k (with possibly less than $W/2$ requests) with another one, k' , thus we have $s_k + s_{k'} > W$. Then it is easy to see that the total number of requests handled by the solution at this level j is greater than $(S_j - 2)\frac{W}{2} + W = S_j\frac{W}{2}$. Moreover, we know that the total number of requests at level j is $\sum_{i \in \cup_{1 \leq k \leq j} \mathcal{C}_k} r_i$, and thus $S_j < \frac{2}{W} \sum_{i \in \cup_{1 \leq k \leq j} \mathcal{C}_k} r_i \leq j$ by hypothesis. This concludes the proof.

From the upper bound on the total number of servers S , we can immediately derive the upper bound on the cost, since M must be at least equal to

$\lceil (\sum_{i \in \mathcal{C}} r_i)/W \rceil$ in order to handle all requests. We have $S = S_n \leq 2(\sum_{i \in \mathcal{C}} r_i)/W$ from the previous reasoning, and thus $S \leq 2M$.

5.2 General Trees

The problem becomes more complex in the case of general trees, because several branches of the tree are interacting. However, the transformation procedure to build a *Single* solution can still be performed, dealing successively with each branch of the tree and enforcing a condition on the minimum number of nodes on each branch.

We start by applying the **linear-tree** procedure successively on each branch of the tree. The number of servers is not guaranteed anymore, thus the procedure may fail. If at the end of the procedure on a branch, some clients have not been assigned to any server, let j be the server which corresponds to the first branching. We consider all requests coming from other branches already assigned to servers j up to the root server, and if possible we either create a new server in the corresponding branch, or move these requests down into this branch. The example of Fig. 4(a) illustrates this procedure. Assume that the two leftmost branches have already been processed, with server 1 handling clients 1 and 9, server 2 handling clients 3 and 6, and server B handling client 2. It is not possible to process requests of clients 7 and 4, thus we move down other requests, either client 9 onto server A (server creation), or client 6 onto server B .

However, it might fail even if the constraint on the minimum number of servers at each level of each branch is respected, as for instance in the example of Fig. 4(b). In this case, each server A,B,C and D will be in charge of the corresponding client with 10 requests, and other requests cannot be assigned to servers 1, 2 and 3 with a *Single* policy.

Thus we add an extra constraint on the minimum number of nodes in a subtree, in order to guarantee the construction of the *Single* solution. Of course, note that the procedure can be applied even if the constraints are violated, but in such cases it might fail. The idea consists in adding a constraint similar to the linear tree one, but generalized on subtrees to allow correct branching. For each node $j \in \mathcal{N}$ such that $|\text{children}(j) \cap \mathcal{N}| \geq 2$, we request that **subtree**(j) follows

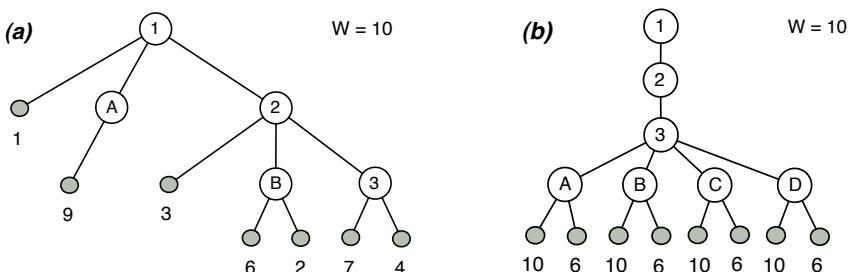


Fig. 4. Dealing with a general tree

the linear tree property. Thus all clients of a branch rooted in j will possibly be processed by servers in $\text{subtree}(j)$, and it will always be possible to process requests when dealing with a new branch.

Formally, the constraint can be written as follows (recall that r is the root of the tree, and that the property is still true for all linear trees starting from the root; this corresponds to the case $j = r$):

$$\forall j \in \{j' \in \mathcal{N} \mid |\text{children}(j') \cap \mathcal{N}| \geq 2\} \cup \{r\}, \forall k \in \text{subtree}(j) \cap \mathcal{N}, \\ \text{let } X = \{j\} \cup \text{ancestors}(k) \cap \text{subtree}(j).$$

$$\text{Then } |X| \geq \frac{2}{W} \sum_{\ell \in X} \sum_{i \in \text{children}(\ell) \cap \mathcal{C}} r_i \quad (1)$$

The algorithm is then the following:

1. For each node $j \in \mathcal{N}$, let $br(j) = |\text{children}(j) \cap \mathcal{N}|$ be the number of branches rooted in j and not yet processed.
2. Call the **linear-tree** procedure on the leftmost branch of the tree, $r = 1 \rightarrow 2 \rightarrow \dots \rightarrow k$. We denote the current branch by $cb = (1, 2, \dots, k) \subseteq \mathcal{N}$, and we mark this branch as processed: $\forall \ell \in cb, br(\ell) = br(\ell) - 1$.
3. For $j = \max_{j' \in cb} \{j' \mid br(j') \geq 1\}$ (first branching from the bottom of the tree), call the **linear-tree** procedure on the leftmost branch rooted in j and not yet processed, denoted as $j \rightarrow j_1 \rightarrow \dots \rightarrow j_k$, with $j < j_1 < \dots < j_k$. We set $cb = (j, j_1, \dots, j_k)$ and mark it as processed: $\forall \ell \in cb, br(\ell) = br(\ell) - 1$.
4. If required, call the **merge-servers** procedure on the current branch.
5. Go back to step 3, until there is no more branching not yet processed (i.e., $\forall j \in \mathcal{N}, br(j) \leq 0$).

We prove in the following that the algorithm returns a *Single* solution with a cost $S \leq 2M$, with in some cases the extra constraint that the tree is binary. At the end of step 2, there is at most one server which is handling less than $W/2$ requests in the branch. The allocation always is possible because of constraint (II) applied on r . For step 3, let us check that there exists a *Single* solution. The number of requests of clients attached to node j_1 is at most W because of constraint (II), thus it is always possible to create a server at this node and handle these requests. Similarly, there is no problem when assigning requests from clients attached to nodes j_2 to j_k . However, at the end of this call, we might have two servers handling less than $W/2$ requests, one in the subtree already processed, x , and one in the new branch, y . In this case, we call the **merge-servers** procedure which aims at suppressing one of these servers. If $x \in \text{ancestors}(j)$ (recall that j is the root of the current branch), then we can simply move requests processed by y to server x , and there remains at most one server with less than $W/2$ requests. Similarly, if one node in $\text{ancestors}(j)$ is not yet a server, we can move requests processed by x and y onto this node, thus merging both servers into a single one. Otherwise, because of constraint (II) at node j , it is always possible to process requests of the current branch without using server j . However, we need at this point an additional constraint on the

subtree rooted in j in order to have the guarantee: it should be a binary tree. With this extra constraint, we can process all requests of both subtrees without using server j , and we can move requests processed by x to server j , since the requests coming from clients attached to j are less than $W/2$ (constraint (II)). Then, if there are still two servers with less than $W/2$ requests, these servers are y and j , and it is possible to move requests from y to j .

Without the binary tree constraint, it might not be possible to merge servers. For instance, consider a tree whose root r has four children nodes, numbered from 1 to 4. Each of these nodes (children of r) has one single client with four requests, and $W = 10$. Then two clients will be processed by r , but there will remain two servers each processing only $4 < W/2$ requests, with no possibility of merging. Note however that the solution still has the performance guarantee $S \leq 2M$, and even in this case $S = M$ since three servers are required for both policies.

Therefore, for a binary tree respecting constraint (II), we can build a *Single* solution with a cost $S \leq 2M$. These constraints can however be relaxed and we expect the procedure to return efficient *Single* solutions in most cases anyway.

6 Related Work

Early work on replica placement by Wolfson and Milo [6] has shown the impact of the write cost and motivated the use of a minimum spanning tree to perform updates between the replicas. In this work, they prove that the replica placement problem in a general graph is NP-complete, and thus they address the case of special topologies, and in particular tree networks. They give a polynomial solution in a fully homogeneous case, using the *Single* closest server access policy. More recent works [2,3,4] use the same access policy, and in each case, the optimization problems are shown to have polynomial complexity. However, the variant with bidirectional links is shown NP-complete by Kalpakis et al [1]. Indeed in [1], requests can be served by any node in the tree, not just the nodes located in the path from the client to the root. All papers listed above consider the *Single* closest access policy. As already stated, most problems are NP-complete, except for some very simplified instances. Karlsson et al [7,8] compare different objective functions and several heuristics to solve these complex problems.

To the best of our knowledge, there is no related work comparing different access policies, either on tree networks or on general graphs. Most previous works impose the closest policy. The *Multiple* policy is enforced by Rodolakis et al [9] but in a very different context: they consider general graphs instead of trees, so they face the combinatorial complexity of finding good routing paths. Also, they assume an unlimited capacity at each node, and they include some QoS constraints in their problem formulation, based on the round trip time (in the graph) required to serve the client requests. In such a context, this (very particular) instance of the *Multiple* problem is shown to be NP-hard.

As already mentioned, we previously tackled these policies with a full complexity study in [5], and Rehn-Sonigo extended these results when communication links are subject to bandwidth constraints, see [10].

7 Conclusion

In this work, we have carefully analyzed different strategies for replica placement, and proved that a *Multiple* solution may be arbitrarily better than a *Single* one, even on homogeneous platforms. Moreover, we have provided an algorithm to build a *Single* solution, which is guaranteed to use no more than two times more servers than the optimal *Multiple* solution, given some constraints on the problem instance. This is a very interesting result, given that the *Single* problem on homogeneous platforms is NP-difficult, and that some applications may not support multiple servers.

Even though the constraints on the trees are quite restrictive, the procedure can be applied on any tree, even if the tree does not allow for a guarantee on the solution. We expect that the ratio of 2 should be achievable in most practical situations. It would be very interesting to simulate the procedure on random application trees and practical ones, in order to figure out the percentage of success and the average performance ratio lost by moving from a *Multiple* solution to a *Single* one. We plan to explore such directions in future work, together with a study in a dynamic setting in which client requests and server capacities evolve over time.

References

1. Kalpakis, K., Dasgupta, K., Wolfson, O.: Optimal placement of replicas in trees with read, write, and storage costs. *IEEE Trans. Parallel and Distributed Systems* 12(6), 628–637 (1985)
2. Cidon, I., Kutten, S., Soffer, R.: Optimal allocation of electronic content. *Computer Networks* 40, 205–218 (2002)
3. Liu, P., Lin, Y.F., Wu, J.J.: Optimal placement of replicas in data grid environments with locality assurance. In: International Conference on Parallel and Distributed Systems (ICPADS). IEEE Computer Society Press, Los Alamitos (2006)
4. Wu, J.J., Lin, Y.F., Liu, P.: Optimal replica placement in hierarchical data grids with locality assurance. *Journal of Parallel and Distributed Computing* 68(12), 1517–1538 (2008)
5. Benoit, A., Rehn-Sonigo, V., Robert, Y.: Replica Placement and Access Policies in Tree Networks. *IEEE Transactions on Parallel and Distributed Systems* 19(12), 1614–1627 (2008)
6. Wolfson, O., Milo, A.: The multicast policy and its relationship to replicated data placement. *ACM Trans. Database Syst.* 16(1), 181–205 (1991)
7. Karlsson, M., Karamanolis, C., Mahalingam, M.: A framework for evaluating replica placement algorithms. Research Report HPL-2002-219, HP Laboratories, Palo Alto, CA (2002)
8. Karlsson, M., Karamanolis, C.: Choosing Replica Placement Heuristics for Wide-Area Systems. In: ICDCS 2004: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS 2004), Washington, DC, USA, pp. 350–359. IEEE Computer Society, Los Alamitos (2004)
9. Rodolakis, G., Siachalou, S., Georgiadis, L.: Replicated server placement with QoS constraints. *IEEE Trans. Par. Distr. Systems* 17(10), 1151–1162 (2006)
10. Rehn-Sonigo, V.: Optimal Closest Policy with QoS and Bandwidth Constraints for Placing Replicas in Tree Networks. In: CoreGRID 2007, Core GRID Symposium 2007. Springer, Heidelberg (2007)

Scheduling Recurrent Precedence-Constrained Task Graphs on a Symmetric Shared-Memory Multiprocessor

UmaMaheswari C. Devi

IBM India Research Laboratory, Bangalore, India

Abstract. We consider approaches that allow task migration for scheduling recurrent directed-acyclic-graph (DAG) tasks on symmetric, shared-memory multiprocessors (SMPs) in order to meet a given throughput requirement with fewer processors. Within the scheduling approach proposed, we present a heuristic based on grouping DAG subtasks for lowering the end-to-end latency and an algorithm for computing an upper bound on latency. Unlike prior work, the purpose of the grouping here is not to map the subtask groups to physical processors, but to generate aggregated entities, each of which can be treated as a single schedulable unit to lower latency. Evaluation using synthetic task sets shows that our approach can lower processor needs considerably while incurring only a modest increase in latency. In contrast to the work presented herein, most prior work on scheduling recurrent DAGs has been for distributed-memory multiprocessors, and has therefore mostly been concerned with statically mapping DAG subtasks to processors.

1 Introduction

Symmetric, shared-memory multiprocessor (SMP) systems, including those based on multicore processors, are now mainstream. With Moore's law manifesting in the form of increasing number of processing cores per chip (as opposed to faster individual cores), only applications with parallelism and to which the available processing resources can be allocated intelligently are likely to witness processing speed-ups. Some applications that can benefit from the current and emerging SMPs include image and digital signal processing systems, deep packet inspection in computer networks, etc.

Several applications like those specified above can be modeled as *directed-acyclic-graphs* (DAGs), also referred to as *task graphs*. In a task graph, nodes denote *subtasks*, and edges, the order in which subtasks should execute on a given input. A simple DAG with six nodes, whose weights denote their execution times, is shown in Fig. 1. Path branches and merges in the DAG are assumed to have fork and join semantics; branches therefore provide *spatial concurrency*. External inputs to some of these systems are recurrent, arriving periodically, such as sampled radar pulses fed into an embedded signal processing system at a specified rate. The external input rate dictates the throughput that should be met. In some systems, processing of later inputs may overlap that of earlier ones, allowing subtasks to be pipelined and enabling *temporal concurrency*. A deadline may also be imposed on the end-to-end latency incurred in processing any given input. Scheduling such recurrent DAGs with spatial and temporal parallelism on SMPs is the subject of this paper.

The problem of scheduling DAGs on multiprocessors has received considerable attention. Much of the early work was limited to scheduling a single DAG instance, while some later and recent work deals with recurring instances. This later work is however focused on distributed-memory multiprocessor (DMM) platforms, on which inter-processor communication and migration costs can be significant. Hence, most of prior work statically maps subtasks onto processors and executes each subtask on its assigned processors only.

Inter-processor communication and migration costs are, in general, less of an issue on SMPs with uniform memory access times, and, in particular, almost negligible on multicore processors with shared caches [6]. Therefore, techniques proposed in the context of DMMs may be overkill for SMPs and need reevaluation for SMPs. An example of the latter class of processors is Sun's eight-core UltraSPARC T1 Niagara chip, whose cores all share a common L2 cache. Also, the number of cores per chip is expected to increase with passing years. In light of these developments, we consider a more flexible scheduling approach in which a given instance or different instances of a subtask may execute on different processors, as opposed to statically binding a subtask to a processor.

Contributions. First, we draw upon optimal multiprocessor rate-based scheduling algorithms from the real-time scheduling literature and let subtasks of a DAG to migrate to minimize the processor needs on an SMP while meeting a stipulated throughput. E.g., in Fig. 1, if external inputs arrive at T once every three time units, then static mapping of subtasks will require at least *six* processors to keep up with the incoming requests. On the other hand, if subtasks can migrate, then they may be scheduled on only *four* processors such that each subtask executes for at least two time units in intervals spanning three time units, which is sufficient to guarantee stability. In this example, processor needs are higher by 50% if migrations are disallowed. On the other hand, lowering the number of processors has the effect of worsening T 's end-to-end latency from six to nine time units. Hence, the approach may not be applicable to all systems, but, resource savings can be quite considerable if some increase in latency can be tolerated. E.g., in systems that consist of multiple tasks as T , each task could contribute to saving two processors, leading to substantial cumulative savings.

Next, within the scheduling approach proposed, we consider lowering end-to-end latency and present a heuristic for the purpose. Our heuristic is based on grouping consecutive subtasks on a sub-path into a task chain that may be scheduled as a single unit. Unlike prior work, the purpose of our grouping is not to determine a mapping from subtasks to processors. Thirdly, we provide an algorithm for computing an upper bound on the end-to-end latency for any input through the DAG under the scheduling approach and heuristic proposed. Finally, we evaluate the efficacy of our methods using synthetic task graphs.

Organization. The rest of this paper is organized as follows. Our task and system model are presented in Sec. 2 followed by a description of the basic scheduling

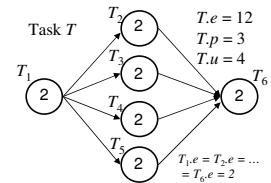


Fig. 1. Sample task graph T

approach that minimizes processor needs in Sec. 3. Heuristic for lowering latency and an algorithm to compute an upper bound on latency are presented in Sec. 4. Sec. 5 provides a simulation-based evaluation, Sec. 6 reviews related work, while Sec. 7 concludes.

2 Task and System Model

We consider scheduling *tasks* modeled as DAGs, also referred to as *task graphs*, on SMPs. Each node of a DAG denotes a *subtask*, which constitutes a sequential section of an entire task. Tasks are denoted using upper-case letters; subtasks are indexed and denoted, *e.g.*, as T_1, T_2, \dots . Since a subtask is sequential, it may not execute on more than one processor at a time although execution on different processors at different times is allowed. Nodes are weighted, with the weight of a node T_i denoting the associated subtask's *worst-case execution cost* $T_i.e$.

A DAG's edges impose a partial order among its nodes, indicating the orders in which subtasks may execute while processing a single request. Branches in paths have fork semantics, hence, all branches should be executed, and may proceed concurrently, if enough resources are available. Inter-processor communication costs are assumed to be negligible, so edges are not weighted. An example task graph with six nodes T_1-T_6 is shown in Fig. 1.

A task is recurrent and triggered by external inputs or *arrivals*. We assume that any two consecutive arrivals to a task T are separated by at least its minimum inter-arrival time, referred to as its *period*, denoted $T.p$. This can be accomplished in practical implementations by delaying an early arrival until its stipulated arrival time (assuming that the arrival rate over the long run does not exceed $1/period$). A task's period may be smaller than the weighted length of any of the paths in its graph, necessitating that subtasks be pipelined. However, a single subtask cannot have more than one instance executing at any time, which we refer to as *subtask concurrency constraint* (SCC). Therefore, a task's period $T.p$ may not be smaller than the execution costs of any of its subtasks T_i , *i.e.* The j^{th} instance of task T and subtask T_i processing the j^{th} arrival are denoted T^j and T_i^j , respectively.

The sum of the execution costs of all the subtasks of T is referred to as the *total execution cost* of T , denoted $T.e$. The ratio of $T.e$ to $T.p$ is the *utilization*, $T.u$, of T . Task preemption and migration costs are assumed to be negligible.

3 Basic Task Graph Scheduling Approach for SMPs

One trivial way of meeting a task T 's throughput requirement, while respecting the SCC, is to order its subtasks in a linear chain that is consistent with the DAG's partial order, round robin external arrivals among available processors, and process each arrival exclusively on a single processor. E.g., in Fig. 1, $T.p=3$, $T.u=4$, so T can be scheduled on four processors by running instance $T^{4 \cdot i+j}$ on processor j for all $i \geq 0$ and $j = 1, 2, 3, 4$. That is, subtasks $T_1^1-T_1^6$ will be run in order on the first processor, $T_2^1-T_2^6$ on the second processor, and so on. It is easy to show that, by this method, it suffices to allocate $\lceil T.u \rceil$ processors to T .

The above method is optimal for task graphs that are linear chains with integral utilization, but not for arbitrary DAGs or chains with non-integral utilization. Also, end-to-end latency could be extremely high for arbitrary DAGs. E.g., latency for T in Fig. 11 is twelve by this approach, and would increase with additional nodes added to the concurrent stage, whereas the length of the longest path would remain at six. In this sense, the approach *does not scale* with increasing concurrency.

In what follows, we propose a scheduling approach that scales with concurrency. For this, we draw upon optimal multiprocessor rate-based scheduling algorithms. We begin with a brief overview.

3.1 Overview of Rate-Based Scheduling on Multiprocessors

Algorithms based on *proportionate-fair* or *Pfair scheduling* [8] are the only known way of *optimally* scheduling systems of stand-alone *rate-based tasks*, which can be thought of as a generalization of *sporadic tasks*, on multiprocessors. A sporadic task T is *sequential*, characterized by a period $T.p$ and a worst-case execution cost $T.e \leq T.p$. T may be invoked or *released* zero or more times, with any two consecutive invocations separated by at least $T.p$ time units. Each invocation of T is referred to as its *job*, is associated with a deadline that equals its period, and should complete executing within $T.p$ time units of its release. The ratio $T.e/T.p \leq 1.0$ denotes T 's utilization $T.u$, which can be thought of as the *rate* at which T executes. Pfair algorithms are optimal for scheduling a task set τ of sporadic or rate-based tasks on M identical processors in that no job of any task would miss its deadline if the total utilization of τ is at most M .

Pfair algorithms achieve optimality by breaking each task into uniform-sized quanta and scheduling tasks one quantum at a time. Processor time is also allocated to tasks in discrete units of quanta. As such, all references to time will be non-negative integers. The time interval $[t, t + 1)$, where t is a non-negative integer, is referred to as slot t . (Hence, time t refers to the beginning of slot t .) The interval $[t_1, t_2)$ consists of slots $t_1, t_1 + 1, \dots, t_2 - 1$.

Each job of T consists of $T.e$ quanta. A task's quanta are numbered contiguously across its jobs (starting from one), thus, quanta $(j - 1) \times T.e + 1$ through $j \times T.e$ comprise the j^{th} job T^j . The i^{th} quantum of T is denoted Q_T^i , and is associated with a release time and deadline, denoted $r(Q_T^i)$ and $d(Q_T^i)$, respectively, which define the window within which it should be scheduled. Release time and deadline for the i^{th} quantum that belongs to the j^{th} job of T (that is, $j = \lceil \frac{i}{T.e} \rceil$) are defined as follows.

$$r(Q_T^i) = r(T^j) - (j - 1) \times T.p + \left\lfloor \frac{i - 1}{T.u} \right\rfloor \wedge d(Q_T^i) = r(T^j) - (j - 1) \times T.p + \left\lceil \frac{i}{T.u} \right\rceil \quad (1)$$

The windows of all the quanta that belong to the same job T^j are contained within the job's window, which spans $[r(T^j), d(T^j))$. Therefore, scheduling each quantum within its window (while ensuring that no two quanta of a task are scheduled concurrently) is sufficient to ensure that all job deadlines of all tasks are met. Windows of consecutive quanta are either disjoint or overlap in one time slot only. Refer to Fig. 2.

A *rate-based* task (also referred to as *intra-sporadic* task) generalizes a sporadic task by allowing quanta within a single job too to be delayed. Such delays are accommodated by shifting quantum windows to the right (by adding the delay to the formulas in (II)) as needed. Refer to the quantum windows of the third job in Fig. 2.

On M processors, Pfair scheduling algorithms function by choosing at the beginning of each time slot, at most M eligible quanta of different tasks for execution in that time slot. Quanta are prioritized by their deadlines, with ties, if any, resolved using non-trivial tie-breaking rules. The most efficient of known Pfair algorithms has a per-slot time complexity of $O(M \log N)$, where N is the number of tasks [8]. Tasks may migrate under Pfair algorithms, and unless migration is allowed, optimality cannot be guaranteed.

3.2 Pfair Scheduling of DAG Tasks

A DAG task T can be Pfair scheduled by treating each of its subtasks as a stand-alone task of a sporadic task system and assigning the stand-alone task the subtask's execution cost and the DAG's period. Subtask dependencies imposed by the DAG can be taken care of by releasing the j^{th} instance T_i^j of subtask T_i only when both of the following hold: (i) the j^{th} instances of all the subtasks that T_i is dependent upon, as well as the previous instance, T_i^{j-1} , of T_i have completed execution, and (ii) at least $T.p$ time units have elapsed since the release time of the previous instance T_i^{j-1} of T_i . The optimality of Pfair scheduling immediately ensures that on $\lceil T.u \rceil$ processors, each instance of each subtask completes within $T.p$ time units of its release, providing the bound in the following theorem on the end-to-end latency through a DAG.

Theorem 1. A DAG task T with total utilization $T.u \leq M$ can be Pfair scheduled on M identical processors such that its end-to-end latency is at most $S \times T.p$, where S is the number of subtasks in the longest, unweighted path in T .

Since the end-to-end latency bound under Pfair depends only on the longest (unweighted) path to any leaf node, adding more nodes without increasing the path length, which amounts to increasing concurrency, would not worsen the latency bound. Hence, this approach can be said to *scale* with concurrency. On the other hand, every unit increase in the path length would, by the above theorem, worsen the latency bound by the DAG's period. This would certainly not be desirable if node execution costs are much less than the period. We address the issue of controlling the increase in latency with increasing path length in the next section. Our approach is based on grouping consecutive nodes on a sub-path into a single schedulable entity. Towards that end, we first consider scheduling linear chains of subtasks.

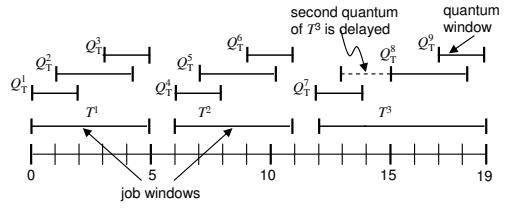


Fig. 2. First three jobs of a rate-based task T with $T.e = 3$ and $T.p = 5$. Job and quantum windows are depicted. Releases of the second and third jobs are delayed by one slot each. The second quantum in the third job is delayed by two slots.

3.3 Scheduling Task Chains

If the subtasks of a linear chain are Pfair scheduled as described in Sec. 3.2, then by Thm. 1, end-to-end latency for the chain could be up to the number of subtasks times the chain's period, which could be high in comparison to the chain's total execution cost. E.g., the subtasks in the chains in Fig. 3 could be Pfair-scheduled on two processors. However, the end-to-end chain latencies could be up to 21 and 14, respectively, which are a little too high.

It is quite easy to see that the two subtasks of V can be combined into a single sporadic task with an execution cost of two and a period of seven that replaces the subtasks and is scheduled as a single entity by a first-level scheduler. Allocations to V can internally be passed on to its two subtasks. Since V is guaranteed to receive two quanta within seven slots of each release, its end-to-end latency bound can be halved to seven.

The total utilization of T is $\frac{12}{7}$, which exceeds 1.0; hence, it is not clear whether the above straightforward approach of grouping a chain's subtasks applied to V extends to T . Scheduling chains with utilization greater than 1.0 is considered next.

Scheduling a task chain with utilization exceeding 1.0. We propose a two-level hierarchical approach for scheduling such task chains. Let S be a task chain with $S.u > 1.0$. Let $S.u^I = \lfloor S.u \rfloor$ and $S.u^f = S.u - S.I < 1.0$ denote the integral and fractional parts of $S.u$, respectively. (For T above, $T.u^I = 1$ and $T.u^f = \frac{5}{7}$.) In our approach, S is assigned $S.u^I + 1$ “fictitious” rate-based tasks (FTs) or virtual processors, $S.u^I$ of which are of unit capacity, and the final one, which is a fractional fictitious task (FT^{fr}, denoted $S.F$), has a utilization $1 \geq S.F.u \geq S.u^f$. FTs of S shall be scheduled along with stand-alone and other FTs by a first-level Pfair scheduler. A second-level scheduler local to S schedules the subtasks of S upon the time allocated to its FTs, while also controlling when the jobs and quanta of $S.F$ are released. For instance, $S.F$'s job may have to be postponed when the chain's release is postponed. Local scheduling within S shall be *preemptive* and prioritize any *ready* subtask that corresponds to an earlier instance over another that corresponds to a later instance. Then, one might expect that end-to-end latency for S should be bounded if $S.F.u = S.u^f$. However, it turns out that, for latency to be bounded, a slightly higher capacity, in the form of utilization greater than $S.u^f$ for their FT^{fr} $S.F$, might be required for some chains S . In what follows, we provide an example of a chain for which no extra capacity is needed and a counterexample for which the minimal capacity does not suffice. Let $T_{i,j}^k$ denote the j^{th} quantum of the k^{th} instance of subtask T_i .

Example. Let T of Fig. 3 be allocated two FTs of utilizations 1.0 and $\frac{5}{7}$, respectively. Allocations due to the first FT are guaranteed every slot, whereas those due to the second whenever that task is scheduled by the first-level scheduler. It can be verified that if the task chain's releases are separated by *exactly* seven slots, then a pattern in which each instance receives five quanta in the first seven slots of its release and thereafter executes continuously until completion, with no allocation going unused, emerges. By this approach, the end-to-end latency for T is lowered to 14 from 21.

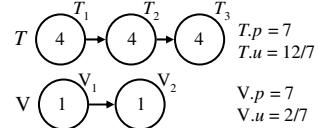


Fig. 3. Sample task chains

Counterexample. This seemingly effective approach can fail if allocations due to FT^{fr} cannot be put to use due to SCC. Consider Y with $Y.u = \frac{4}{3}$ in Fig. 4. The first schedule in the figure is when the capacity allocated to Y is $4/3$; hence, $Y.F.u = 1/3$. Each quantum of $Y.F$ can receive its allocation in any one of the three slots that span its window. If the allocations are always such that they cannot be put to use, as it is the case in the schedule depicted (solid lines in the second row), then latency for Y would grow without bound. In the example, $Y_{1,1}^5$ cannot be scheduled at time 13 because the previous instance of Y_1 , Y_1^4 , is executing at that time. It can be verified that delaying $Y.F$'s quanta do not prove to be effective either.

Bounding latency. One way of ensuring bounded latency for a task as above is to inflate the capacity allocated to it. The increase depends on the task parameters, and in most cases, is much less than rounding to the next integer. For Y above, inflating by $\frac{1}{6}$ from $1\frac{1}{3}$ to $1\frac{1}{2}$ suffices. A schedule for Y with the inflated allocation is also shown in Fig. 4. The extra capacity, if any, needed by a chain is determined in Thm. 2.

To see how inflating helps, note that in our example, when the fractional capacity is $\frac{1}{3}$, it is not possible to guarantee that allocations due to the FT^{fr} do not occur when a prior instance of the first subtask in the chain executes. For instance, if the third quantum due to $Y.F$ were available at time 6 or 7 (instead of 8) in the first schedule, then $Y_{1,1}^3$ could have executed concurrently with $Y_{2,2}$. Increasing the capacity to $\frac{1}{2}$ gives us the flexibility to postpone the release of the fractional task when wastage is possible, so that at least one quantum that can be put to use gets allocated within three slots of a release of Y .

Before we proceed further, some notation is in order. For any task chain C , let $C.e^f \stackrel{\text{def}}{=} C.e\%C.p$, and $C.e_i \stackrel{\text{def}}{=} \sum_{k=1}^i C_k.e$, the cumulative execution cost of its first i subtasks. Take $C.e_0 = 0$. Let $C.\sigma$ denote that subtask of C that would have commenced execution but not completed if C were allocated $C.e^f$ time slots. Formally, $C.\sigma \stackrel{\text{def}}{=} C_k$, where $k = (i | C.e_{i-1} < C.e^f \wedge C.e_i > C.e^f)$, if an i as defined exists, and \emptyset , otherwise. Finally, let $C.\sigma.e^1$ denote the number of quanta of $C.\sigma$ that are contained in the initial $C.e^f$ quanta of C , and $C.\sigma.e^2$, the remaining number of quanta of the same subtask. (If $C.\sigma = \emptyset$, then $C.\sigma.e^1 = C.\sigma.e^2 = 0$.) For T in Fig. 3, $T.e^f = 5$, $T.\sigma = T_2$, $T.\sigma.e^1 = 1$ and $T.\sigma.e^2 = 3$.

Since subtasks are prioritized by the instances they are part of, of all the pending instances, the latest instance of C , C^ℓ , is bound to receive the least allocation in the $C.p$ slots of its release. Our goal is to ensure that this allocation to C^ℓ is at least $C.e\%C.p$ quanta, by ensuring that at least this many quanta due to $C.F$ remain “usable.”

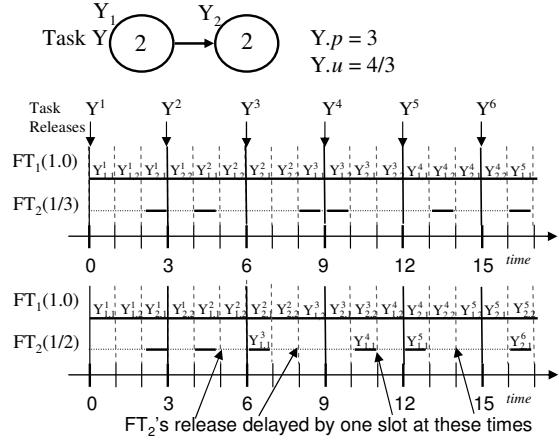


Fig. 4. Initial segments of schedules for the task chain Y when allocated processing capacities of $1\frac{1}{3}$ and $1\frac{1}{2}$

Before we proceed further, some notation is in order. For any task chain C , let $C.e^f \stackrel{\text{def}}{=} C.e\%C.p$, and $C.e_i \stackrel{\text{def}}{=} \sum_{k=1}^i C_k.e$, the cumulative execution cost of its first i subtasks. Take $C.e_0 = 0$. Let $C.\sigma$ denote that subtask of C that would have commenced execution but not completed if C were allocated $C.e^f$ time slots. Formally, $C.\sigma \stackrel{\text{def}}{=} C_k$, where $k = (i | C.e_{i-1} < C.e^f \wedge C.e_i > C.e^f)$, if an i as defined exists, and \emptyset , otherwise. Finally, let $C.\sigma.e^1$ denote the number of quanta of $C.\sigma$ that are contained in the initial $C.e^f$ quanta of C , and $C.\sigma.e^2$, the remaining number of quanta of the same subtask. (If $C.\sigma = \emptyset$, then $C.\sigma.e^1 = C.\sigma.e^2 = 0$.) For T in Fig. 3, $T.e^f = 5$, $T.\sigma = T_2$, $T.\sigma.e^1 = 1$ and $T.\sigma.e^2 = 3$.

Since subtasks are prioritized by the instances they are part of, of all the pending instances, the latest instance of C , C^ℓ , is bound to receive the least allocation in the $C.p$ slots of its release. Our goal is to ensure that this allocation to C^ℓ is at least $C.e\%C.p$ quanta, by ensuring that at least this many quanta due to $C.F$ remain “usable.”

If an instance of C , C^j , receives $C.e^f$ quanta in the first $C.p$ time slots of its release, then $C.\sigma^j$ would receive $C.\sigma.e^1$ of those, and the amount of execution still pending for $C.\sigma^j$ would be $C.\sigma.e^2$. If C^j can execute continuously until completion after $C.p$ slots, then $C.\sigma^j$ would complete in as many slots as it has pending quanta. Hence, our goal would be accomplished if the allocations needed for the $C.\sigma.e^1$ initial quanta of the next instance of subtask $C.\sigma$, $C.\sigma^{j+1}$, are guaranteed to be available in $C.p - C.\sigma.e^2$ slots following the completion of $C.\sigma^j$. As is formally proved below, this is possible if the processing capacity allocated to C is increased to the maximum of $C.u^f$ and $C.u^{f'} \stackrel{\text{def}}{=} \frac{C.\sigma.e^1}{C.p - C.\sigma.e^2}$.

Theorem 2. *A task chain C with $C.u > 1.0$ can be guaranteed an end-to-end latency bound of $\lceil C.u \rceil \times C.p$ if C is allocated a processing capacity due to at least $\lfloor C.u \rfloor$ unit capacity FTs and one FT^{fr} , $C.F$, with utilization $C.F.u = \max(\frac{C.e^f}{C.p}, \frac{C.\sigma.e^1}{C.p - C.\sigma.e^2})$.*

Proof. Let local scheduling within C be as described earlier. Let $n = \lceil C.u \rceil - \lfloor C.u \rfloor$. We prove the theorem by claiming that if a capacity as specified is allocated, then every instance of C executes (a) for at least $C.e^f \leq C.p$ slots in the first $n \times C.p$ slots of its release and (b) continuously thereafter until completion. The bound on latency would immediately follow. (Part (b) of the claim assumes that each instance of each subtask executes for exactly its execution cost, and each instance of C executes for exactly $C.e^f$ slots in the first $C.p$ slots. This assumption can be relaxed at the cost of a longer proof.)

The proof of the above claim is by induction on C 's instances.

Base Case. Since the first instance of C is prioritized over the remaining instances, it can execute continuously from release until completion. Hence, it would receive all of the initial $C.p$ slots, which forms the base case.

Induction Step. Assume that the claim holds for the first $k-1$ instances (induction hypothesis, IH). We first prove Part (a), *i.e.*, show that the k^{th} instance executes for at least $C.e^f$ slots within $n \times C.p$ slots of its release. If $C.u$ is integral, then this part is vacuously true. Hence, assume otherwise, so $n = 1$. Let t denote the release time of C^k , $r(C^k)$. Because C 's releases are at least $C.p$ slots apart, by IH, no subtask preceding $C.\sigma$ would be pending in any prior instance, and the prior instance of $C.\sigma$, $C.\sigma^{k-1}$, completes execution by time $t + C.\sigma.e^2$. Therefore, it suffices to ensure that the k^{th} instance of $C.\sigma$, $C.\sigma^k$, can receive at least $C.\sigma.e^1$ quanta in $[t + C.\sigma.e^2, t + C.p)$, while the k^{th} instances of the subtasks preceding $C.\sigma$ receive at least $C.e^f - C.\sigma.e^1$ quanta before $C.\sigma^k$. We consider two cases.

Case 1. $\frac{C.e^f}{C.p} \geq \frac{C.\sigma.e^1}{C.p - C.\sigma.e^2}$. In this case, $C.F.u = \frac{C.e^f}{C.p}$ and $C.F.p = C.p$. Therefore, since C 's arrivals are separated by at least $C.p$ slots, $C.F$'s releases can be made to coincide with those of C . Hence, a new instance of $C.F$ is released at t . The number of quanta windows of $C.F$ fully contained in the interval $[t, t + C.p)$ is exactly $C.e^f$, its execution cost. Of these, at most $\lceil C.F.u \times C.\sigma.e^2 \rceil$ are contained (either fully or partially) in $[t, t + C.\sigma.e^2)$ in which $C.\sigma^{k-1}$ may execute. Therefore, the number of windows fully contained in $[t + C.\sigma.e^2, t + C.p)$ is at least $C.e^f - \lceil C.F.u \times C.\sigma.e^2 \rceil$. Since $C.\sigma.e^2 = C.\sigma.e - C.\sigma.e^1$ and $C.F.u = \frac{C.e^f}{C.p}$, we have

$$\# \text{ of quantum windows fully contained in } [t + C.\sigma.e^2, t + C.p) \geq C.e^f - \lceil \frac{C.e^f \times (C.\sigma.e - C.\sigma.e^1)}{C.p} \rceil. \quad (2)$$

By $\frac{C.e^f}{C.p} \geq \frac{C.\sigma.e^1}{C.p - C.\sigma.e^2}$, we have $\frac{C.\sigma.e - C.\sigma.e^1}{C.p} \times C.e^f \leq C.e^f - C.\sigma.e^1$ (cross-multiply and use $C.\sigma.e^2 = C.\sigma.e - C.\sigma.e^1$), substituting which in (2), we have the number of windows fully contained in $[t + C.\sigma.e^2, t + C.p]$ to be at least $C.\sigma.e^1$. Therefore, $C.\sigma^k$ can execute for $C.\sigma.e^1$ slots without conflict with $C.\sigma^{k-1}$, while the previous allocations due to $C.F$ can be made use of by the preceding subtasks to execute for $C.e^f - C.\sigma.e^1$ quanta.

Case 2. $\frac{C.e^f}{C.p} < \frac{C.\sigma.e^1}{C.p - C.\sigma.e^2}$. In this case, cross-multiplying and rearranging terms in its condition, we have $\frac{C.e^f}{C.p} > \frac{C.e^f - C.\sigma.e^1}{C.\sigma.e^2}$. Hence, $C.F$'s capacity in this case, $\frac{C.\sigma.e^1}{C.p - C.\sigma.e^2}$, is greater than $\frac{C.e^f - C.\sigma.e^1}{C.\sigma.e^2}$. Also, $C.p = C.\sigma.e^2 + (C.p - C.\sigma.e^2)$. Therefore, to ensure that $C.\sigma^k$ does not become ready until its previous instance $C.\sigma^{k-1}$ completes at $t + C.\sigma.e^2$, $C.F$ can be throttled to receive allocations at a lower rate of $\frac{C.e^f - C.\sigma.e^1}{C.\sigma.e^2}$ in the first $C.\sigma.e^2$ slots of C^k 's release, by releasing its job at t with execution cost $C.e^f - C.\sigma.e^1$ and deadline $t + C.\sigma.e^2$. This would result in an allocation of exactly $C.e^f - C.\sigma.e^1$ quanta in $[t, t + C.\sigma.e^2]$ which can be used by subtasks preceding $C.\sigma^k$. At time $t + C.\sigma.e^2$, when $C.\sigma^{k-1}$ completes execution, $C.F$'s job released at t would also complete. Hence, $C.F$'s capacity can be restored and a new job released for it at that time with execution cost $C.\sigma.e^1$ and deadline $t + C.p - C.\sigma.e^2$, which would lead to an allocation of $C.\sigma.e^1$ quanta for $C.\sigma^k$. Changing a task's utilization at job boundaries, as long as the total utilization of the task system does not exceed the processing capacity available, does not lead to any deadline misses under Pfair scheduling, as proved in [9]. Thus, $C.F$ can be made to receive exactly $C.e^f$ quanta and C^k to execute for $C.e^f$ slots in the first $C.p$ slots.

We are left with proving Part (b). Suppose to the contrary that C^k does not execute continuously from time $t + n \times C.p$. Let t' denote the earliest time at or after $t + n \times C.p$ that C^k does not execute. Then, at t' , either all the processors are busy executing prior instances of C , or some subtask of C^k has its previous instance still executing and hence cannot commence. The former implies that at least $\lfloor C.u \rfloor$ prior instances of C are still executing at t' . The release time of $(k - \lfloor C.u \rfloor)^{\text{th}}$ instance is at or before $t - C.p \times \lfloor C.u \rfloor$, and hence, by IH, completes at or before $t - C.p \times \lfloor C.u \rfloor + \lceil C.u \rceil \times C.p \leq t + C.p \leq t'$, a contradiction. Therefore, C^k 's execution cannot be stalled due to lack of processors. We next show that C^k 's execution is not blocked due to SCC either. Suppose not; let C_ℓ^k be the first subtask in C^k that is blocked because C_ℓ^{k-1} is still executing. Since $r(C^{k-1}) \leq t - C.p$, C_ℓ^{k-1} completes at $t - C.p + (C.e_\ell - C.e^f) + C.p = t + C.e_\ell - C.e^f$ (by IH). The completion time of $C_{\ell-1}^k$ is given by $t + (C.e_{\ell-1} - C.e^f) + C.p$, which, because $C_\ell.e \leq C.p$, is at least $t + (C.e_{\ell-1} - C.e^f) + C_\ell.e$, that is, at least $t + C.e_\ell - C.e^f$, the completion time of C_ℓ^{k-1} . Thus, C_ℓ^{k-1} completes before $C_{\ell-1}^k$, a contradiction again. The theorem follows. ■

Corollary 1. *The end-to-end latency of a task chain when it is Pfair-scheduled as a single entity is at most the latency when its subtasks are Pfair-scheduled independently.*

Proof. Follows from Thms. 1 and 2 because $C.u$ is at most the number of C 's subtasks. ■

4 Managing Latency in Arbitrary DAGs

In this section, we first present a heuristic for lowering end-to-end latency in arbitrary DAGs. The basic idea is to construct one or more (sub-)task chains, each of which consists of one or more consecutive subtasks in a sub-path of a DAG, and is scheduled as a single entity. By Cor. 11, latency through the grouped subtasks is at most that when they are scheduled independently, and is likely to be much lower when grouped well. After dealing with task chain construction, we consider computing a bound on the end-to-end latency.

4.1 Chain Construction Heuristic

The basic idea is to iteratively do *depth-first-search* from candidate nodes (nodes with no incoming edges from subtasks not yet grouped) to identify the next best (sub-)path whose nodes can be grouped. Candidate paths are built such that no node in the path has an incoming edge from a node that is *not* in either the path under construction or a chain constructed in a previous step. This way, the chain sequence would be acyclic, which simplifies latency computation. Some criteria for identifying the best path are inflation to utilization, effectiveness in lowering latency, path length *etc.*

An example is provided in Fig. 5. Grouping starts from T_1 . Two candidate paths from T_1 are $C_1 = T_1T_2T_4$ and $C'_1 = T_1T_3$. These paths cannot be extended further since there are incoming edges to T_6 and T_5 from T_5 and T_2 , respectively, that are not on their respective paths. $C_1.e^f = 5$, $C_1.u^f = \frac{5}{10}$, and $C'_1.u^f = \frac{2}{6}$. $C_1.u^f > C'_1.u^f$, so by Thm. 2, C_1 's utilization need not be inflated. Since $C'_1.u = \frac{7}{10} < 1.0$, no inflation is needed for C'_1 , either. Of the two, we choose the longer path, C_1 . The next candidate node is T_3 , the path $T_3T_5T_6$ from which covers all the remaining nodes. This path too does not need inflation to utilization. It should be noted that if extending a path by one or more nodes leads to higher inflation to utilization, then our heuristic does not perform the extension.

One complication that arises when scheduling a set of chains constructed from a DAG is that any node in a chain (not necessarily the first) may have an incoming edge from any node (not necessarily the last) in a previous chain. Edges (T_1, T_3) and (T_2, T_5) are examples. Consequently, an instance of a chain may block after it commences. E.g., it can be verified that T_3^1 (in C_2) becomes ready at most six time units after T_1^1 's release, and can complete by time 11, whereas T_5^1 (in C_2 again) may not be ready until time 14. If the blocking subtask is served by the FT^{fr} , then during the stall, the pending quanta of the FT^{fr} should be delayed (recall that a rate-based task's quanta can be delayed) to ensure that the allocations due to it do not go waste and latency is bounded. In our example, the release of the 5th quantum of $C_2.F^1$ can be delayed until time 14.

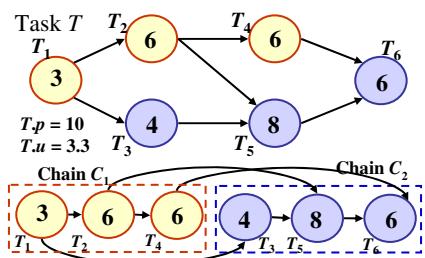


Fig. 5. Grouping subtasks into chains

```

procedure CONSTRUCT-CHAINS (DAG  $T$ )
1 while  $num\_ungrouped\_nodes > 0$  do
2   for each  $candidate\_node$  do
3      $curr\_chain := \text{CHAIN-FROM}$ 
        ( $candidate\_node$ ,
          $\emptyset$ );
4     if  $\text{ISBETTER}(curr\_chain,$ 
         $best\_chain)$  then
5       update  $best\_chain$ ,
         $num\_ungrouped\_nodes$ ,
        list of grouped nodes;
        fi
6     od
6 coalesce as many adjacent chains as
possible into a single one

```

```

procedure CHAIN-FROM(integer  $next\_node$ , CHAIN
 $curr\_chain$ ) returns CHAIN
1 if there is an incoming edge from an ungrouped node to
 $next\_node$  then
2   return  $curr\_chain$ ;
3   else
4     append  $next\_node$  to  $curr\_chain$ ;
        fi
5    $best\_chain := curr\_chain$ ;
6   for each  $node \in adj\_list(next\_node)$  do
7      $curr\_chain := \text{CHAIN-FROM}(node, curr\_chain)$ ;
8     if  $\text{ISBETTER}(curr\_chain, best\_chain)$  THEN
9        $best\_chain := curr\_chain$ ;
        fi
 $od$ 
9 return  $best\_chain$ ;

```

Fig. 6. Heuristic to group subtasks into chains

As can be seen from Sec. 4.2, latency through the DAG after grouping is at most 29 time units, which is only 1.26 times the length of the longest DAG path (23), while requiring no extra processing capacity. In comparison, if subtasks were statically mapped to processors, at least five processors ($>50\%$ extra capacity) would be required, and if subtasks were Pfair-scheduled without grouping, then the end-to-end latency could be up to 40.

Pseudo-code for the chain-construction heuristic is provided in Fig. 6 and should be self-explanatory. After basic chains are constructed, adjacent chains are coalesced if doing so has scope to lower either latency or inflation to utilization. When coalesced, nodes that belonged to different chains need not be on a path in the DAG. In such a case, an imposed dependency edge is introduced as appropriate. The complexity of the heuristic is $O(V(V + E))$, where V is the number of nodes, and E , the number of edges, in the DAG.

4.2 End-to-End Latency Bound

Our latency-computation algorithm for a DAG grouped into chains is based on that for determining the DAG's longest path. Pseudo-code is provided in Fig. 7. Nodes are considered in a topologically-sorted order by considering chains, and nodes within chains, in the order they were included while grouping. For each node, the latest time it would complete is determined by determining the completion time along each incoming edge.

Some aspects to note are as follows. First the time spent in a node can exceed its weight if the node is served by its chain's FT^{fr} . In such a case, the additional time taken can be computed from the release time and deadline (determined using Eq. (1)) of the quanta of the FT^{fr} that will serve it. The accuracy of the end result can be increased if the latest completion time for the node is computed (as opposed to the time spent in it) by collectively considering all preceding nodes in the longest path to the node (through the incoming edge under consideration) that are in the same chain.

E.g., in Fig. 3, the latest completion time for T_2 would be ten if the sub-path consisting of its predecessor T_1 and itself is considered as a single unit. On the other hand, if the nodes are considered individually, and the maximum time through each is determined (using (1)), then the latest completion time for T_2 could be 11, which is a tad loose.

A second aspect is that a sub-task earlier in a chain may be delayed by prior instances of later subtasks in the chain. Such delays should be properly accounted for. E.g., if T in Fig. 5 is released every ten slots, then T_3^3 cannot commence until time 29 (nine slots after T^3 's release) even though it is ready at time 26, since T_6^1 and the first four quanta of T_5^2 may not

complete until time 29. Hence, since T_3 is served by $C_2.F$ and $C_2.F.u = \frac{8}{10}$, and so can require up to five slots for its four quanta, the completion time of an instance of T_3 can be up to 14 slots from T 's release time. If T_3 were to have an outgoing edge

```

procedure LATENCY-BOUND (DAG  $T$ , DAG_chains  $chains$ )
1  for each chain  $C \in chains$  do            $\triangleright$  taken in order
2    for each node of  $C$  do            $\triangleright$  taken in order
3       $l\_e\_time[node] := T_{node}.e$ ;
4       $l\_s\_time[node] := 0$ ;
5      for each incoming-edge into node do
6         $\triangleright$  Compute the first and last quanta in the sub-path of
7         $\triangleright$  all nodes (in  $C$ ) in the longest path to node
8         $\triangleright$  along incoming-edge
9        if incoming-edge is from pred-node in  $C$  then
10           $st\_q := first\_q[pred\_node]$ ;
11           $end\_q := st\_q + path\_e\_in\_chain[pred\_node]$ 
12           $+ T.e_{node} - 1$ ;
13        else
14           $st\_q := T.e_{node} - T_{node}.e + 1$ ;
15           $end\_q := T.e_{node}$ ;
16        fi
17        compute end-time and start-time along
18          incoming-edge using  $st\_q$ ,  $end\_q$ ,  $C.u^f$ ,
19           $C.u^{f'}$ ,  $C.F.u$ ,  $l\_s\_time[pred\_node]$ , and Eq. (1);
20        if end-time >  $l\_e\_time[node]$  then
21           $l\_e\_time[node] := end\_time$ ;
22           $l\_s\_time[node] := start\_time$ ;
23           $first\_q[node] := st\_q$ ;
24           $path\_e\_in\_chain[node] := end\_q - st\_q + 1$ ;
25        fi
26      od
27      if  $l\_e\_time[node] > max\_t$  then  $max\_t := l\_e\_time[node]$ ; fi
28    od
29     $\triangleright$  Adjust end-time to take into account delays due to
30     $\triangleright$  earlier instances of later subtasks
31    for each node of  $C$  do            $\triangleright$  taken in reverse order
32      if  $l\_s\_time[node] > l\_e\_time[pred\_node\_in\_chain]$  then
33        update  $l\_e\_time[pred\_node\_in\_chain]$  and
34         $l\_s\_time[pred\_node\_in\_chain]$  appropriately;
35      fi
36    od
37  od
38  return  $max\_t$ ;

```

Fig. 7. Algorithm to compute end-to-end latency bound through a DAG decomposed into chains

to a later chain, then the ready time of the subtask that the edge is incident to should be taken as 14 and not 11. In Fig. 7 such delays are accounted for by the **for** loop beginning at line 20. It can be shown (by induction over chains) that latency is bounded despite such delays. A formal proof is omitted due to lack of space. The complexity of the algorithm is $O(E + V)$.

5 Empirical Evaluation

Experiments were conducted using randomly-generated task graphs to evaluate the efficacy of Pfair scheduling (with and without grouping) in lowering processor needs and latency.

Random DAGs were generated using the following configuration parameters: total number of DAG nodes (`total_nodes`), the upper limit on spatial concurrency (`max_conc`), and the maximum length (in nodes) of a concurrent path (`conc_len`). Fig. 8 provides an illustration. Nodes were added in the order of stages until the limit of `total_nodes` was reached. The number of nodes in a sub-stage of a concurrent stage was distributed uniformly between $0.6 \times \text{max_conc}$ and `max_conc`. When a new node was added, needed edges connecting it to those already present were inserted. All leaf nodes that existed after the addition of `total_nodes` – 1 were linked to the final node. Cross edges connecting concurrent paths were randomly inserted with a uniform probability of 0.05. Each DAG’s period was set to 20 and subtask execution costs were uniformly distributed between 6 and 15.

For each DAG generated, we determined upper bounds on latencies under Pfair scheduling with and without grouping. We also determined the number of processors that would be needed if migration were disallowed. Results are shown in Fig. 9. Each value reported is the average determined for 10,000 DAGs. Inset (a) shows latency results for DAGs with 31 nodes (resp., 16 nodes) for `max_conc` values of 2, 4, and 8, (resp., 2, 4, and 6) with `conc_len` set to three (resp., two) in each case. Our comparison metric is the ratio of latency under the considered approach to the longest weighted path through the DAG (a lower bound on latency). As expected, latencies are considerably and consistently lower when subtasks are grouped. Also as expected, when the number of nodes is constant, the efficacy of grouping decreases with increasing concurrency, though it is still better by over 30% even in the worst case reported. Turning to the processing needs of the various approaches, a processing capacity that equals the total utilization (whose average value is 17.5 and 8.5 for 31 and 16 nodes, respectively) suffices when nodes are ungrouped. When grouped, the average total inflation to a DAG is around 1% of the total utilization, while it is close to 50% under static mapping. Inset (b) shows the results for constant `max_conc` but varying `total_nodes`, and hence, varying path lengths. Results for other values of configuration parameters were similar, but could differ if subtask execution costs are chosen differently.

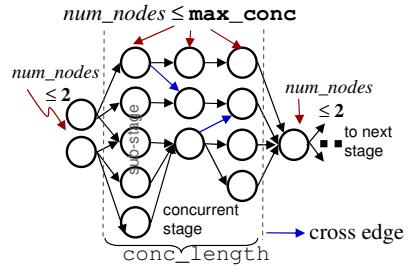


Fig. 8. Random DAG structure

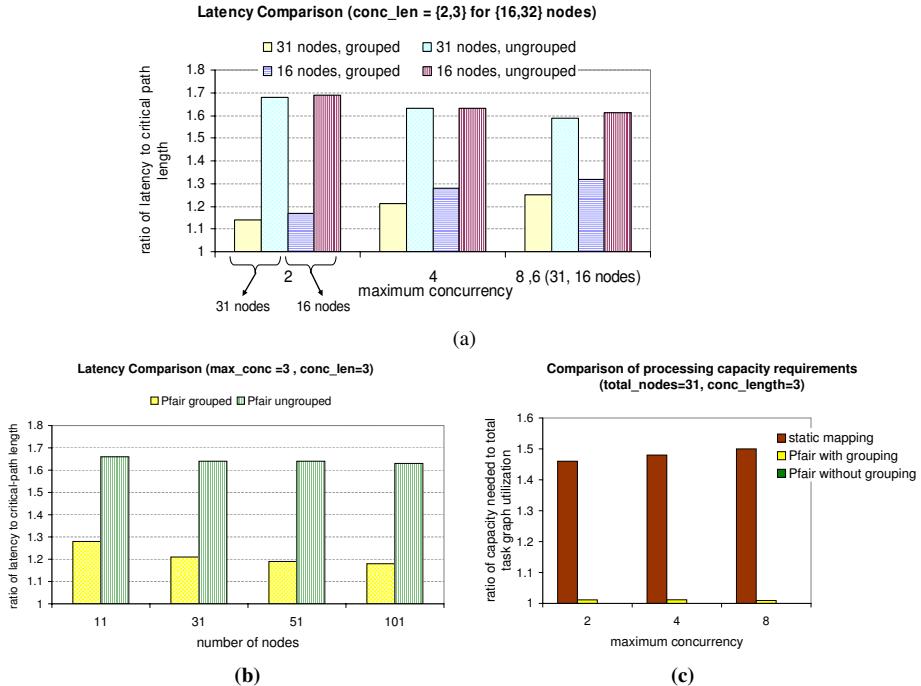


Fig. 9. Comparison of latencies and processor needs under different approaches

6 Related Work

Scheduling a task with concurrency modeled as a DAG on multiprocessors is well-studied, with a large body of work focused on minimizing the makespan of a single task instance. Since the problem is NP-complete for the general version and for most except a few simple variants [2], the focus has been on devising efficient heuristics. Refer to [7] for a survey.

Some later and recent work in this area has turned to scheduling recurring instances of DAG tasks. Most of the work is for DMMs and considers statically mapping tasks to processors under differing assumptions on task concurrency and structure. Scheduling DSP applications to maximize throughput is considered in [5]. The target platform in this work is closely-coupled, but has local processor memories and a segmented bus that make memory access times non-uniform. Optimizing latency under throughput constraint and vice versa by assigning parallelizable subtasks (which thereby allow data parallelism) to multiple processors is considered in [1]. Mapping a chain of data-parallel tasks to processors, including replicating subtasks for optimizing throughput, is the subject of [10], while evaluating latency-throughput tradeoffs that of [11].

There has also been work targeting specific architectures. E.g., scheduling on processors connected by point-to-point networks to meet the throughput requirement is considered in [4], while network of workstations are targeted in [12] for scheduling

video-processing algorithms to optimize the number of processors needed for a desired throughput and vice versa. Heuristics for mapping tasks of streaming applications for execution on workstations of a cluster is considered in [3].

All of the work referred to above is for DMMs in which inter-processor communication and migration overheads can be significant. Hence, applying techniques proposed therein for SMPs can be overkill.

7 Conclusion

We have proposed allowing the subtasks of a DAG task to migrate across the processors or cores of an SMP to enable meeting throughput requirements with fewer processors. We have also proposed a heuristic for lowering the end-to-end latency of a DAG and an algorithm for determining an upper bound on that measure under the scheduling approach proposed. Empirical evaluation using synthetic task graphs shows that our approaches can significantly lower processor needs, while incurring only a modest increase in latency in comparison to those that prohibit migration.

Some avenues for future work are as follows. First, the proposed algorithms can be extended to architectures in which not all but only subsets of cores share common caches and evaluated on a multicore test-bed. Second, latency computation can be incorporated within the grouping heuristic to construct better groups that can lower latency, and guarantees on performance that can be made in general can be determined. Finally, the latency computation algorithm can be extended for bursty arrivals and stochastic workloads.

References

1. Choudhary, A., Narahari, B., Nicol, D., Simha, R.: Optimal processor assignment for a class of pipelined computations. *IEEE Transactions on Parallel and Distributed Systems* 5(4), 439–445 (1994)
2. Garey, M., Johnson, D.: *Computers and Intractability: a Guide to the Theory of NP-Completeness*. W.H. Freeman, New York (1979)
3. Guirado, F., Ripoll, A., Roig, C., Luque, E.: Optimizing latency under throughput requirements for streaming applications on cluster execution. In: *Proceedings of the IEEE International Conference on Cluster Computing*, September 2005, pp. 1–10 (2005)
4. Hary, S., Ozguner, F.: Precedence-constrained task allocation onto point-to-point networks for pipelined execution. *IEEE Transactions on Parallel and Distributed Systems* 10(8), 838–851 (1999)
5. Hoang, P., Rabaey, J.: Scheduling dsp programs onto multiprocessors for maximum throughput. *IEEE Transactions on Signal Processing* 41(6), 2225–2235 (1993)
6. Kazempour, V., Fedorova, A., Alagheband, P.: Performance implications of cache affinity on multicore processors. In: *Proceedings of the 14th International Conference on Parallel Computing*, August 2008, pp. 151–162 (2008)
7. Kwok, Y.K., Ahmad, I.: Static scheduling algorithms for allocating derected task graphs to multiprocessors. *ACM Computing Surveys* 31(4), 406–471 (1999)
8. Srinivasan, A., Anderson, J.: Optimal rate-based scheduling on multiprocessors. In: *Proceedings of the 34th ACM Symposium on Theory of Computing*, May 2002, pp. 189–198 (2002)

9. Srinivasan, A., Anderson, J.: Fair scheduling of dynamic task systems on multiprocessors. *Journal of Systems and Software* 77(1), 67–80 (2005)
10. Subhlok, J., Vondran, G.: Optimal mapping of sequences of data parallel tasks. In: *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, July 1995, pp. 134–143 (1995)
11. Subhlok, J., Vondran, G.: Optimal latency-throughput tradeoffs for data parallel machines. In: *Proceedings of the 8th ACM Symposium on Parallel Algorithms and Architectures*, June 1996, pp. 62–71 (1996)
12. Yang, M.-T., Kasturi, R., Sivasubramaniam, A.: A pipeline-based approach for scheduling video processing algorithms on NOW. *IEEE Transactions on Parallel and Distributed Systems* 14(2), 119–130 (2003)

Energy-Aware Scheduling of Flow Applications on Master-Worker Platforms

Jean-François Pineau⁵, Yves Robert^{2,3,4}, and Frédéric Vivien^{1,3,4}

¹ INRIA

² ENS Lyon

³ Université de Lyon

⁴ LIP laboratory, ENS Lyon–CNRS–INRIA–UCBL, Lyon, France

⁵ LIRMM laboratory, UMR 5506, CNRS–Université Montpellier 2, France

Abstract. We consider the problem of scheduling an application composed of independent tasks on a fully heterogeneous master-worker platform with communication costs. We introduce a bi-criteria approach aiming at maximizing the throughput of the application while minimizing the energy consumed by participating resources. Assuming arbitrary super-linear power consumption laws, we investigate different models for energy consumption, with and without start-up overheads. Building upon closed-form expressions for the uniprocessor case, we derive optimal or asymptotically optimal solutions for both models.

1 Introduction

The Earth Simulator requires about 12 megawatts of peak power, and Petaflop systems may require 100 MW of power, nearly the output of a small power plant (300 MW). At \$100 per MegaWatt.Hour, peak operation of a petaflop machine may thus cost \$10,000 per hour [1]. And these estimates ignore the additional cost of dedicated cooling. Current estimates state that cooling costs \$1 to \$3 per watt of heat dissipated [2]. This is just one of the many economical reasons why energy-aware scheduling is an important issue, even without considering battery-powered systems such as laptop and embedded systems.

Many important scheduling problems involve large collections of identical tasks [3,4]. In this paper, we consider a single bag-of-tasks application which is launched on a heterogeneous platform. We suppose that all processors have a discrete number of speeds (or modes) of computation: the quicker the speed, the less efficient energetically-speaking. Our aim is to maximize the throughput, i.e., the fractional number of tasks processed per time-unit, while minimizing the energy consumed. Unfortunately, the goals of low power consumption and efficient scheduling are contradictory. Indeed, throughput can be maximized by using more energy to speed up processors, while energy can be minimized by reducing the speeds of the processors, hence the total throughput.

Altogether, power-aware scheduling truly is a bi-criteria optimization problem. A common approach to such problems is to fix a threshold for one objective

and to minimize the other. This leads to two interesting questions. If we fix energy, we get the *laptop problem*, which asks “What is the best schedule achievable using a particular energy budget, before battery becomes critically low?”. Fixing schedule quality gives the *server problem*, which asks “What is the least energy required to achieve a desired level of performance?”.

The contribution of this work is to consider a fully heterogeneous master-worker platform, and to take communication costs into account. Here is the summary of our main results:

- Under an ideal power-consumption model, we derive an optimal polynomial algorithm to solve either bi-criteria problem (maximize throughput within a power consumption threshold, or minimize energy consumption while guaranteeing a required throughput).
- Under a refined power-consumption model with start-up overheads, we derive a polynomial algorithm which is asymptotically optimal.

This paper is organized as follows. We first present the framework and different power consumption models in Section 2. We study the bi-criteria scheduling problem under the ideal power consumption model in Section 3, and under the more realistic model with overheads in Section 4. Section 5 is devoted to an overview of related work. Finally, we state some concluding remarks in Section 6.

2 Framework

We outline in this section the model for the target applications and platforms, as well as the characteristics of the consumption model. Next we formally state the bi-criteria optimization problem.

2.1 Application and Platform Model

We consider a bag-of-tasks application \mathcal{A} , composed of a large number of independent, same-size tasks, to be deployed on a heterogeneous master-worker platform. We let ω be the amount of computation (expressed in *flops*) required to process a task, and δ be the volume of data (expressed in *bytes*) to be communicated for each task. We do not consider return messages. This simplifying hypothesis could be alleviated by considering longer messages (append the return message for a given task to the incoming message of the next one).

The master-worker platform, also called star network, or single-level tree in the literature, is composed of a master P_{master} , the root of the tree, and p workers P_u ($1 \leq u \leq p$). Without loss of generality, we assume that the master has no processing capability. Otherwise, we can simulate the computations of the master by adding an extra worker paying no communication cost. The link between P_{master} and P_u has a bandwidth b_u . We assume a linear cost model: it takes a time δ/b_u to send a task to processor P_u . We suppose that the master can send/receive data to/from all workers at a given time-step according to the *bounded multi-port* model [5,6]. There is a limit on the total amount of data that

the master can send per time-unit, denoted as BW. Intuitively, the bound BW corresponds to the bandwidth capacity of the master's network card; the flow of data out of the card can be either directed to a single link or split among several links, hence the multi-port hypothesis. We also assume a computation model called *synchronous start* computation: the computation of a task on a worker can start at the same time as the reception of the task starts, provided that the computation rate is no greater than the communication rate (the communication must complete before the computation). This models the fact that, in several applications, only the first bytes of data are needed to start executing a task. In addition, the theoretical results of this paper are more easily expressed under this model, which provides an upper bound on the achievable performance. However, results in [7] show that proofs written under that model can be extended to more realistic models (one-port communication and atomic computation).

2.2 Energy Model

Among the main system-level energy-saving techniques, Dynamic Voltage Scaling (DVS) works on a very simple principle: decrease the supply voltage (and so the clock frequency) to the CPU so as to consume less power. For this reason, DVS is also called *frequency-scaling* or *speed scaling* [8]. We suppose a discrete voltage-scaling model. The computational speed of worker P_u has to be picked among a limited number of m_u modes. We denote the computational speeds $s_{u,i}$, meaning that processor P_u running in the i th mode (noted $P_{u,i}$) needs $\omega/s_{u,i}$ time-units to execute one task of \mathcal{A} . We suppose that processing speeds are listed in increasing order ($s_{u,1} \leq s_{u,2} \leq \dots \leq s_{u,m_u}$), and modes are exclusive: one processor can only run in a single mode at any given time.

Rather than assuming a relation of the form $P_d = s^\alpha$ where P_d is the power dissipation, s the processor speed, and α some constant greater than 1, we adopt a more general approach, as we only assume that power consumption is a *super-linear* function (i.e., a strictly increasing and convex function) of the processor speed. We denote by $\mathfrak{P}_{u,i}$ the power consumption per time unit of processor $P_{u,i}$. We focus on two power consumption models. Under the **ideal model**, switching among the modes does not cost any penalty, and an idle processor does not consume any power. Consequently, for each processor P_u , the power consumption is super-linear from 0 to the power consumption at frequency $s_{u,1}$. Under the **model with start-up overheads**, once a processor is on, its power consumption is non-null and at least that of its idle frequency, or speed, $s_{u,1}$. If the cost of turning on and off a processor is null, this model is meaningless. This is why we need to add a start-up overhead. Under this more realistic model, power consumption now depends on the duration of the interval during which the processor is turned on (the overhead is only paid once during this interval). We introduce a new notation to express power consumption as a function of the length t of the execution interval:

$$\mathfrak{P}_{u,i}(t) = \mathfrak{P}_{u,i}^{(1)} \cdot t + \mathfrak{P}_u^{(2)} \quad (1)$$

where $\mathfrak{P}_u^{(2)}$ is the energy overhead to turn processor P_u on.

To summarize, we consider two models: an **ideal model** simply characterized by $\mathfrak{P}_{u,i}$, the power consumption per time-unit of P_u running in mode i , and a **model with start-up overheads**, where power consumption is given by Equation 1 for each processor.

2.3 Objective Function

Our goal is bi-criteria scheduling: the first objective is to minimize the power consumption, and the second to maximize the throughput. We denote by $\rho_{u,i}$ the throughput of worker $P_{u,i}$ for application \mathcal{A} , i.e., the average number of tasks $P_{u,i}$ executes per time-unit. There is a limit to the number of tasks that each mode of one processor can perform per time-unit. First of all, as $P_{u,i}$ runs at speed $s_{u,i}$, it cannot execute more than $s_{u,i}/\omega$ tasks per time-unit. Second, as P_u can only be at one mode at a time, and $\frac{\rho_{u,i} \omega}{s_{u,i}}$ represents the fraction of time spent under mode $m_{u,i}$ per time-unit, this constraint can be expressed by:

$$\forall u \in [1..p], \sum_{i=1}^{m_u} \frac{\rho_{u,i} \omega}{s_{u,i}} \leq 1.$$

Under the ideal model, and for the simplicity of proofs, we can add an additional idle mode $P_{u,0}$ whose speed is $s_{u,0} = 0$. As the power consumption per time-unit of $P_{u,i}$, when fully used, is $\mathfrak{P}_{u,i}$ ($\mathfrak{P}_{u,0} = 0$), its power consumption per time-unit with a throughput of $\rho_{u,i}$ is then $\frac{\rho_{u,i} \omega}{s_{u,i}} \mathfrak{P}_{u,i}$.

We denote by ρ_u the throughput of worker P_u , i.e., the sum of the throughput of each mode of P_u (except the throughput of the idle mode), so the total throughput of the platform is denoted by:

$$\rho = \sum_{u=1}^p \rho_u = \sum_{u=1}^p \sum_{i=1}^{m_u} \rho_{u,i}.$$

We define problem **MINPOWER** (ρ) as the problem of minimizing the power consumption while achieving a throughput ρ . Similarly, **MAXTHROUGHPUT** (\mathfrak{P}) is the problem of maximizing the throughput while not exceeding the power consumption \mathfrak{P} . In Section 3 we deal with the ideal model. We extend this work to a more realistic model in Section 4.

3 Ideal Model

Both bi-criteria problems (maximizing the throughput given an upper bound on power consumption and minimizing the power consumption given a lower bound on throughput) have been studied at the processor level, using particular power consumption laws such as $P_d = s^\alpha$ [9][10][11]. However, we solve these problems optimally using the sole assumption that the power consumption is super-linear. Furthermore, we solve these problems at the platform level, that

is, for a heterogeneous set of processors. A key step is to establish closed-form formulas linking power consumption and throughput on a single processor:

Proposition 1. *For any processor P_u , the optimal power consumption to achieve a throughput of $\rho > 0$ is*

$$\mathfrak{P}_u(\rho) = \max_{0 \leq i < m_u} \left\{ (\omega\rho - s_{u,i}) \frac{\mathfrak{P}_{u,i+1} - \mathfrak{P}_{u,i}}{s_{u,i+1} - s_{u,i}} + \mathfrak{P}_{u,i} \right\},$$

and is obtained using two consecutive modes, P_{u,i_0} and P_{u,i_0+1} , such that $\frac{s_{u,i_0}}{\omega} < \rho \leq \frac{s_{u,i_0+1}}{\omega}$.

The following result shows how to solve the converse problem, namely maximizing the throughout subject to a prescribed bound on power consumption.

Proposition 2. *The maximum achievable throughput according to the power consumption limit \mathfrak{P} is*

$$\rho_u(\mathfrak{P}) = \min \left\{ \frac{s_{u,m_u}}{\omega}; \max_{1 \leq i \leq m_u} \left\{ \frac{\mathfrak{P}(s_{u,i+1} - s_{u,i}) + s_{u,i}\mathfrak{P}_{u,i+1} - s_{u,i+1}\mathfrak{P}_{u,i}}{\omega(\mathfrak{P}_{u,i+1} - \mathfrak{P}_{u,i})} \right\} \right\},$$

and is obtained using two consecutive modes, P_{u,i_0} and P_{u,i_0+1} , such that: $\mathfrak{P}_{u,i_0} < \mathfrak{P} \leq \mathfrak{P}_{u,i_0+1}$.

Due to lack of space, see [7] for the proofs.

To the best of our knowledge, these uni-processor formulas, linking the throughput to the power consumption, are new, even for standard laws. They will prove to be very useful when dealing with multi-processor problems.

3.1 Minimizing Power Consumption

Thanks to Propositions 1 and 2, we do not need to specify the throughput for each frequency on any given processor. We only have to fix a throughput for each processor to know how to achieve the minimum power consumption on that processor. Furthermore, the bounded multi-port hypothesis is easy to take into account: either the outgoing capacity of the master is able to ensure the given throughput ($\text{BW} \geq \rho$), or the system has no solution. Overall, we have the following linear program (Equation 2). This linear program is defined by three types of constraints: the first constraint states that the system has to ensure the given throughput; the second set of constraints states that the processing capacity of a processor P_u as well as the bandwidth of the link from P_{master} to P_u are not exceeded; the last constraint links the power consumption of one processor according to its throughput.

$$\left\{
\begin{array}{l}
\text{MINIMIZE } \mathfrak{P} = \sum_{u=1}^p \mathfrak{P}_u \text{ SUBJECT TO} \\
\sum_{u=1}^p \rho_u = \rho \\
\forall u, \rho_u \leq \min \left\{ \frac{s_{u,m_u}}{\omega}; \frac{b_u}{\delta} \right\} \\
\forall u, \forall 1 \leq i \leq m_u, \mathfrak{P}_u \geq (\omega \rho_u - s_{u,i}) \frac{\mathfrak{P}_{u,i+1} - \mathfrak{P}_{u,i}}{s_{u,i+1} - s_{u,i}} + \mathfrak{P}_{u,i}
\end{array}
\right. \quad (2)$$

For each value \mathfrak{P}_u used in the objective function (recall that \mathfrak{P}_u is the power consumption per time unit of P_u), we have m_u equations (see Proposition 11). When looking at the constraints, we observe that the problem can be optimally solved using a greedy strategy. We first sort processors in an increasing order according to their power consumption ratio. This power consumption ratio depends on the different modes of the processors, and the same processor will appear a number of times equal to its number of modes. Formally, we sort in non decreasing order the quantities $\left\{ \frac{\mathfrak{P}_{u,i+1} - \mathfrak{P}_{u,i}}{s_{u,i+1} - s_{u,i}} \right\}$. The next step is to select the cheapest mode of the processors so that the system can achieve the required throughput, given that each processor throughput is limited by its maximal frequency and the bandwidth of the link between itself and the master. Altogether, we obtain Algorithm 11.

Algorithm 1. Greedy algorithm minimizing power consumption under a given throughput

Data: throughput ρ that has to be achieved

for $u = 1$ **to** p **do**

$T[u] \leftarrow 0$; /* throughput of processor P_u */

$\Phi \leftarrow 0$; /* total throughput of the system */

$\mathcal{L} \leftarrow$ sorted list of the P_{u_k, i_k} such that $\forall j$,

$$\frac{\mathfrak{P}_{u_j,1+i_j} - \mathfrak{P}_{u_j,i_j}}{s_{u_j,1+i_j} - s_{u_j,i_j}} \leq \frac{\mathfrak{P}_{u_{j+1},1+i_{j+1}} - \mathfrak{P}_{u_{j+1},i_{j+1}}}{s_{u_{j+1},1+i_{j+1}} - s_{u_{j+1},i_{j+1}}};$$

while $\Phi < \rho$ **do**

$P_{u_k, i_k} \leftarrow \text{next}(\mathcal{L})$; /* selection of next cheapest mode */

$\rho' \leftarrow T[u_k]$; /* previous throughput of P_{u_k} (at mode $i_k - 1$) */

$T[u_k] \leftarrow \min \left\{ \frac{s_{u_k, i_k}}{\omega}; \frac{b_{u_k}}{\delta}; \rho' + (\rho - \Phi) \right\}$; /* new throughput of P_{u_k} (at mode i_k) */

if $T[u_k] = \frac{b_{u_k}}{\delta}$ **then**

$\mathcal{L} \leftarrow \mathcal{L} \setminus \{P_{u_k, j}\}$; /* no need to look at faster modes for P_{u_k} */

$\Phi \leftarrow \Phi + T[u_k] - \rho'$;

One can detail more precisely the line labeled /* new throughput */ that gives the new throughput of P_{u_k} at mode i_k . This throughput is bounded by the maximum throughput at this speed, by the maximum communication

throughput, and also by the previous throughput (ρ') plus the remaining throughput that has to be achieved ($\rho - \Phi$). We point out that, if the last selected mode is $P_{u_{k_0}, i_{k_0}}$, Algorithm 1 will

1. fully use each processor having at least one mode consuming strictly less than $P_{u_{k_0}, i_{k_0}}$, and this either at the throughput of the bandwidth if reached (this throughput is achieved according to Proposition 1), or at the largest single fastest mode that consumes strictly less than $P_{u_{k_0}, i_{k_0}}$ or at the same mode than $P_{u_{k_0}, i_{k_0}}$;
2. either not use at all or fully use at its first non-trivial mode any processor whose first non-trivial mode consumes exactly the same as $P_{u_{k_0}, i_{k_0}}$;
3. not use at all any processor whose first non-trivial mode consumes strictly more than the mode $P_{u_{k_0}, i_{k_0}}$;
4. use $P_{u_{k_0}, i_{k_0}}$ at the minimum throughput so the system achieves a throughput of ρ (according to Proposition 1).

Theorem 1. *Algorithm 1 optimally solves problem MINPOWER (ρ) (see linear program (2)).*

Due to lack of space, see [7] for the proof.

3.2 Maximizing the Throughput

Maximizing the throughput is a very similar problem. We only need to adapt Algorithm 1 so that the objective function considered during the selection process is replaced by the power consumption:

$$T[u_k] \leftarrow \min \left\{ \mathfrak{P}_{u_k, i_k}; \left(\omega \frac{b_{u_k}}{\delta} - s_{u_k, i_k} \right) \frac{\mathfrak{P}_{u_k, i_k+1} - \mathfrak{P}_{u_k, i_k}}{s_{u_k, i_k+1} - s_{u_k, i_k}} + \mathfrak{P}_{u_k, i_k}; \mathfrak{P}' + (\mathfrak{P} - \Psi) \right\}$$

where Ψ is the current power consumption (we iterate while $\Psi \leq \mathfrak{P}$). The proof that this modified algorithm **optimally solves** problem MAXTHROUGHPUT (\mathfrak{P}) is very similar to that of Algorithm 1 and can be found in [7].

4 Model with Start-Up Overheads

When we move to more realistic models, the problem gets much more complicated. In this section, we still look at the problem of minimizing the power consumption of the system with a throughput bound, but now we suppose that there is a power consumption overhead when turning a processor on. We denote this problem MINPOWEROVERHEAD (ρ). First we need to modify the closed-form formula given by Proposition 1, in order to determine the power consumption of processor P_u when running at throughput ρ_u during t time-units. The new formula is then:

$$\begin{aligned}
\mathfrak{P}_u(t, \rho_u) &= \max_{0 \leq i < m_u} \left\{ (\omega \rho_u - s_{u,i}) \frac{\mathfrak{P}_{u,i+1}(t) - \mathfrak{P}_{u,i}(t)}{s_{u,i+1} - s_{u,i}} + \mathfrak{P}_{u,i}(t) \right\} \\
&= \max_{0 \leq i < m_u} \left\{ (\omega \rho_u - s_{u,i}) \frac{\mathfrak{P}_{u,i+1}^{(1)} - \mathfrak{P}_{u,i}^{(1)}}{s_{u,i+1} - s_{u,i}} \cdot t + \mathfrak{P}_{u,i}^{(1)} \cdot t \right\} + \mathfrak{P}_u^{(2)} \\
&= \mathfrak{P}_u^{(1)}(\rho_u) \cdot t + \mathfrak{P}_u^{(2)}.
\end{aligned}$$

The overhead is paid only once, and the throughput ρ_u is still obtained by using the same two modes P_{u,i_0} and P_{u,i_0+1} as in Proposition 1. We first run the mode P_{u,i_0} during $\frac{t(s_{u,i_0+1} - \rho_u \omega)}{s_{u,i_0+1} - s_{u,i_0}}$ time-units, then the mode P_{u,i_0+1} during $\frac{t(\rho_u \omega - s_{u,i_0})}{s_{u,i_0+1} - s_{u,i_0}}$ time-units (these values are obtained from the fraction of time the modes are used per time-unit; see [7] for more details). We can now prove the following dominance property about optimal schedules (see [7] for the proof):

Proposition 3. *There exists an optimal schedule in which all processors, except possibly one, are used at a maximum throughput, i.e., either the throughput dictated by their bandwidth, or the throughput achieved by one of their execution modes.*

Unfortunately, Proposition 3 does not help design an optimal algorithm. However, a modified version of the previous algorithm remains asymptotically optimal. The general principle of the approach is as follows: instead of looking at the power consumption per time-unit, we look at the energy consumed during d time-units, where d will be later defined. Let α_u be the throughput of P_u during d time-units. Thus, the throughput of each processor per time-unit is $\rho_u = \frac{\alpha_u}{d}$. As all processors are not necessarily enrolled, let \mathcal{U} be the set of the selected processors' indexes. The constraint on the energy consumption can be written:

$$\begin{aligned}
\forall u, \forall 1 \leq i \leq m_u, \mathfrak{P}_u \cdot d &\geq \left((\omega \rho_u - s_{u,i}) \frac{\mathfrak{P}_{u,i+1} - \mathfrak{P}_{u,i}}{s_{u,i+1} - s_{u,i}} + \mathfrak{P}_{u,i} \right) \cdot d + \mathfrak{P}_u^{(2)}, \\
\text{or, } \forall u, \forall 1 \leq i \leq m_u, \mathfrak{P}_u - \frac{\mathfrak{P}_u^{(2)}}{d} &\geq (\omega \rho_u - s_{u,i}) \frac{\mathfrak{P}_{u,i+1} - \mathfrak{P}_{u,i}}{s_{u,i+1} - s_{u,i}} + \mathfrak{P}_{u,i}.
\end{aligned}$$

The linear program is then:

$$\left\{
\begin{aligned}
&\text{MINIMIZE } \mathfrak{P} = \sum_{u \in \mathcal{U}} \mathfrak{P}_u \text{ SUBJECT TO} \\
&\sum_{u=1}^p \rho_u = \rho \\
&\forall u, \rho_u \leq \min \left\{ \frac{s_{u,m_u}}{\omega}, \frac{b_u}{\delta} \right\} \\
&\forall u \in \mathcal{U}, \forall 1 \leq i \leq m_u, \mathfrak{P}_u - \frac{\mathfrak{P}_u^{(2)}}{d} \geq (\omega \rho_u - s_{u,i}) \frac{\mathfrak{P}_{u,i+1} - \mathfrak{P}_{u,i}}{s_{u,i+1} - s_{u,i}} + \mathfrak{P}_{u,i}
\end{aligned}
\right. \tag{3}$$

However, this linear program cannot be solved unless we know \mathcal{U} . So we need to add some constraints. In the meantime, we make a tiny substitution into the objective function ($\mathfrak{P}'_u = \mathfrak{P}_u - \frac{\mathfrak{P}_u^{(2)}}{d}$), in order to simplify the last constraint (the first two constraints remain unchanged):

$$\left\{ \begin{array}{l} \text{MINIMIZE } \mathfrak{P} = \sum_{u=1}^p \left(\mathfrak{P}'_u + \frac{\mathfrak{P}_u^{(2)}}{d} \right) \text{ SUBJECT TO} \\ \dots \\ \forall u, \forall 1 \leq i \leq m_u, \mathfrak{P}'_u \geq (\omega \rho_u - s_{u,i}) \frac{\mathfrak{P}_{u,i+1} - \mathfrak{P}_{u,i}}{s_{u,i+1} - s_{u,i}} + \mathfrak{P}_{u,i} \end{array} \right. \quad (4)$$

The inequalities are stronger than previously, so every solution of (4) is a solution of (3). Of course, optimal solutions for (4) are most certainly not optimal for the initial problem (3). However, the larger d , the closer the constraints are from each other. Furthermore, Algorithm II builds optimal solutions for (4). So, the expectation is that when d becomes large, solutions built by Algorithm II becomes good approximate solutions for (4). Indeed we derive the following result (see [7] for the proof):

Theorem 2. *Algorithm II is asymptotically optimal for problem MINPOWER-OVERHEAD (ρ) (see linear program (3)).*

5 Related Work

Several papers have been targeting the minimization of power consumption. Most of them suppose they can switch to arbitrary speed values.

- **Unit time tasks.** Bunde in [11] focuses on the problem of offline scheduling unit time tasks with release dates, while minimizing the makespan or the total flow time on one processor. He chooses to have a continuous range of speeds for the processors. He extends his work from one processor to multi-processors, but unlike this paper, does not take any communication time into account. He also proves the NP-completeness of the problem of minimizing the makespan on multi-processors with jobs of different amount of work. Authors in [9] concentrate on minimizing the total flow time of unit time jobs with release dates on one processor. After proving that no online algorithm can achieve a constant competitive ratio if job have arbitrary sizes, they exhibit a constant competitive online algorithm and solve the offline problem in polynomial time. Contrarily to [11] where tasks are gathered into blocks and scheduled with increasing speed in order to minimize the makespan, here the authors prove that the speed of the blocks need to be decreasing in order to minimize both total flow time and the energy consumption.
- **Communication-aware.** In [12], the authors are interested about scheduling task graphs with data dependencies while minimizing the energy consumption of both the processors and the inter-processor communication

devices. They demonstrate that in the context of multiprocessor systems, the inter-processor communications were an important source of consumption, and their algorithm reduces up to 80% the communications. However, as they focus on multiprocessor problems, they only consider the energy consumption of the communications, and they suppose that the communication times are negligible compared to the computation times.

- **Discrete voltage case.** In [13], the authors deal with the problem of scheduling tasks on a single processor with discrete voltages. They also look at the model where the energy consumption is related to the task, and describe how to split the voltage for each task. They extend their work in [14] to online problems. In [15], the authors add the constraint that the voltage can only be changed at each cycle of every task, in order to limit the number of transitions and thus the energy overhead. They find that under this model, the minimal number of frequency transitions in order to minimize the energy may be greater than two.
- **Task-related consumption.** [16] addresses the problem of periodic independent real-time tasks on one processor, the period being a deadline to all tasks. In this work the energy consumption is related to the *task* that is executed on the processor. A polynomial algorithm is exhibited to find the optimal speed of each task, and it is shown that EDF can be used to obtain a feasible schedule with these optimal speed values.
- **Deadlines.** In [17], the authors focus on the problem where tasks arrive according to some release dates. They show that during any elementary time interval defined by some release dates and deadlines of applications, the optimal voltage is constant, and they determine this voltage, as well as the minimum constant speed for each job. [10] improves the best known competitive ratio to minimize the energy while respecting all deadlines. [18] works with an overloaded processor (which means that no algorithm can finish all the jobs) and try to maximize the throughput. Their online algorithm is $O(1)$ competitive for both throughput maximization and energy minimization. [19] has a similar approach by allowing task rejection, and proves the NP-hardness of the studied problem.
- **Slack sharing.** In [20,21], the authors investigate dynamic scheduling. They consider the problem of scheduling DAGs before deadlines, using a semi-clairvoyant model. For each task, the only information available is the worst-case execution time. Their algorithm operates in two steps: first a greedy static algorithm schedules the tasks on the processors according to their worst-case execution times and the deadline, and reduces the processors speed so that each processor meets the deadline. Then, if a task ends sooner than according to the static algorithm, a dynamic slack sharing algorithm uses the extra-time to reduce the speed of computations for the following tasks. However, they do not take communications into account.
- **Heterogeneous multiprocessor systems.** Authors in [22] study the problem of scheduling real-time tasks on two heterogeneous processors. They provide a FPTAS to derive a solution very close to the optimal energy consumption with a reasonable complexity. In [23], the authors propose a greedy

algorithm based on affinity to assign frame-based real-time tasks, and then they re-assign them in pseudo-polynomial time when any processing speed can be assigned for a processor. Authors of [24] propose an algorithm based on integer linear programming to minimize the energy consumption without guarantees on the schedulability of a derived solution for systems with discrete voltage. Some authors also explored the search of approximation algorithms for the minimization of allocation cost of processors under energy constraints [25,26].

6 Conclusion

In this paper, we have studied the problem of scheduling a single application with power consumption constraints, on a heterogeneous master-worker platform. We derived new closed-form relations between the throughput and the power consumption at the processor level. These formulas enabled us to develop an optimal bi-criteria algorithm under the ideal power consumption model.

Moving to a more realistic model with start-up overheads, we were able to prove that our approach provides an asymptotically optimal solution. We hope that our results will provide a sound theoretical basis for forthcoming studies.

As future work, it would be interesting to address sophisticated models with frequency switching costs, which we expect to lead to NP-hard optimization problems, and then look for some approximation algorithms.

Acknowledgment. This work was supported in part by the ANR StochaGrid project.

References

1. Ge, R., Feng, X., Cameron, K.W.: Performance-constrained distributed DVS scheduling for scientific applications on power-aware clusters. In: Proceedings of the 2005 ACM/IEEE conference on Supercomputing (SC 2005). IEEE CS, Los Alamitos (2005)
2. Skadron, K., Stan, M.R., Sankaranarayanan, K., Huang, W., Velusamy, S., Tarjan, D.: Temperature-aware microarchitecture: Modeling and implementation. ACM Transactions on Architecture and Code Optimization 1(1), 94–125 (2004)
3. Casanova, H., Berman, F.: Parameter Sweeps on the Grid with APST. In: Hey, A., Berman, F., Fox, G. (eds.) Grid Computing: Making The Global Infrastructure a Reality. John Wiley, Chichester (2003)
4. Adler, M., Gong, Y., Rosenberg, A.L.: Optimal sharing of bags of tasks in heterogeneous clusters. In: Proceedings of SPAA, pp. 1–10. ACM Press, New York (2003)
5. Hong, B., Prasanna, V.: Distributed adaptive task allocation in heterogeneous computing environments to maximize throughput. In: Proceedings of IPDPS. IEEE CS, Los Alamitos (2004)
6. Hong, B., Prasanna, V.K.: Adaptive allocation of independent tasks to maximize throughput. IEEE TPDS 18(10), 1420–1435 (2007)
7. Pineau, J.F.: Communication-aware scheduling on heterogeneous master-worker platforms. PhD thesis, ENS Lyon (2008)

8. Hotta, Y., Sato, M., Kimura, H., Matsuoka, S., Boku, T., Takahashi, D.: Profile-based optimization of power performance by using dynamic voltage scaling on a PC cluster. In: Proceedings of IPDPS. IEEE CS, Los Alamitos (2006)
9. Albers, S., Fujiwara, H.: Energy-efficient algorithms for flow time minimization. *ACM Transactions on Algorithms* 3(4) (2007)
10. Bansal, N., Kimbrel, T., Pruhs, K.: Dynamic speed scaling to manage energy and temperature. In: Foundations of Computer Science (FoCS), pp. 520–529 (2004)
11. Bunde, D.P.: Power-aware scheduling for makespan and flow. In: Proceedings of SPAA, pp. 190–196. ACM Press, New York (2006)
12. Varatkar, G., Marculescu, R.: Communication-aware task scheduling and voltage selection for total systems energy minimization. In: International Conference on Computer-Aided Design (ICCAD). IEEE CS, Los Alamitos (2003)
13. Ishihara, T., Yasuura, H.: Voltage scheduling problem for dynamically variable voltage processors. In: Proceedings of ISLPED, pp. 197–202. ACM Press, New York (1998)
14. Okuma, T., Ishihara, T., Yasuura, H.: Real-time task scheduling for a variable voltage processor. In: Proceedings of ISSS. IEEE CS, Los Alamitos (1999)
15. Zhang, Y., Hu, X.S., Chen, D.Z.: Energy minimization of real-time tasks on variable voltage processors with transition energy overhead. In: Asia South Pacific Design Automation Conference (ASPDAC), pp. 65–70. ACM Press, New York (2003)
16. Aydin, H., Melhem, R., Mosse, D., Mejia-Alvarez, P.: Determining optimal processor speeds for periodic real-time tasks with different power characteristics. In: Proceedings of EMRTS, pp. 225–232. IEEE CS, Los Alamitos (2001)
17. Quan, G., Hu, X.: Energy efficient fixed-priority scheduling for real-time systems on variable voltage processors. In: Design Automation Conference, pp. 828–833 (2001)
18. Chan, H.L., Chan, W.T., Lam, T.W., Lee, L.K., Mak, K.S., Wong, P.W.H.: Energy efficient online deadline scheduling. In: Proceedings of SODA, pp. 795–804. SIAM, Philadelphia (2007)
19. Chen, J.J., Kuo, T.W., Yang, C.L., King, K.J.: Energy-efficient real-time task scheduling with task rejection. In: Proceedings of DATE, European Design and Automation Association, pp. 1629–1634 (2007)
20. Zhu, D., Melhem, R., Childers, B.R.: Scheduling with dynamic voltage/speed adjustment using slack reclamation in multiprocessor real-time systems. *IEEE TPDS* 14(7), 686–700 (2003)
21. Rusu, C., Melhem, R., Mossé, D.: Multi-version scheduling in rechargeable energy-aware real-time systems. *Journal of Embedded Computing* 1(2), 271–283 (2005)
22. Chen, J.-J., Thiele, L.: Energy-efficient task partition for periodic real-time tasks on platforms with dual processing elements. In: Proceedings of ICPADS. IEEE CS, Los Alamitos (2008)
23. Huang, T.Y., Tsai, Y.C., Chu, E.H.: A near-optimal solution for the heterogeneous multi-processor single-level voltage setup problem. In: Proceedings of IPDPS (2007)
24. Yu, Y., Prasanna, V.: Power-aware resource allocation for independent tasks in heterogeneous real-time systems. In: Proceedings of ICPADS, pp. 341–348 (2002)
25. Chen, J.J., Kuo, T.W.: Allocation cost minimization for periodic hard real-time tasks in energy-constrained dvs systems. In: International Conference on Computer-Aided Design (ICCAD), pp. 255–260. ACM, New York (2006)
26. Hsu, H.-R., Chen, J.-J., Kuo, T.-W.: Multiprocessor synthesis for periodic hard real-time tasks under a given energy constraint. In: Proceedings of DATE, pp. 1061–1066. European Design and Automation Association (2006)

Topic 4

High Performance Architectures and Compilers

Introduction

Pedro C. Diniz*, Ben Juurlink*, Alain Darte*, and Wolfgang Karl*

This topic deals with architecture design and compilation for high performance systems. The areas of interest range from microprocessors to large-scale parallel machines; from general-purpose platforms to specialized hardware (e.g., graphic coprocessors, low-power embedded systems); and from hardware design to compiler technology. On the compilation side, topics of interest include programmer productivity issues, concurrent and/or sequential language aspects, program analysis, transformation, automatic discovery and/or management of parallelism at all levels, and the interaction between the compiler and the rest of the system. On the architecture side, the scope spans system architectures, processor micro-architecture, memory hierarchy, and multi-threading, and the impact of emerging trends.

Out of the 17 papers submitted to this topic, 4 were accepted for presentation at the conference, corresponding to an acceptance rate of 24%. The submissions to this topic as well as the accepted papers highlight the growing significance of Chip Multi-Processors (CMP) and Simultaneous Multi-Threaded (SMT) processors in contemporary high-performance architectures.

The paper “Last Bank: Dealing with Address Reuse in Non-Uniform Cache Architectures for CMPs”, by Javier Lira, Carlos Molina and Antonio Gonzalez, proposes the use of an additional memory bank, the Last Bank, in non-uniform cache architectures (NUCA). The authors explore two policies for using this last bank that acts as a last-level cache and deals with data that has been evicted from other banks and can thus be saved from leaving the chip.

The paper “Paired ROBs: A Cost-Effective Reorder Buffer Sharing Strategy for SMT Processors”, by Rafael Ubal Tena, Julio Sahuillo, Salvador Petit, and Pedro Lopez, deals with the sharing of instruction ROBs (Reorder Buffers) in simultaneous multi-thread processors. The authors present experimental results that reveal that for light-load degrees of concurrency the proposed strategy of organizing the ROB in pairs outperforms the common approach of private ROBs.

The paper “REPAS: Reliable Execution for Parallel Applications in Tiled-CMPs” by Daniel Sánchez, Juan L. Aragón, and José M. García, deals with the growing concern of reliability. This paper explores the use of a novel redundant multi-threading (RMT) using a limited amount of hardware resources in simultaneous multi-threaded cores where a large portion of hardware resources is shared. Their results for several parallel scientific benchmarks reveal that the

* Topic Chairs.

proposed technique provides full coverage against soft-errors with a lower performance slowdown in comparison to non-redundant systems.

Last but not the least, the paper “Impact of Quad-core Cray XT4 System and Software Stack on Scientific Computation” by S. Alam, R. F. Barrett, H. Jagode, J. A. Kuehn, S. W. Poole, and R. Sankaran, focuses on the evaluation of key architectural and software features on the execution of large-scale scientific applications. The author provides an analysis of different strategies in the tuning of applications exploring vectorization instructions, deeper memory hierarchy, SIMD instructions, and a multicore-aware MPI library.

Last Bank: Dealing with Address Reuse in Non-Uniform Cache Architecture for CMPs

Javier Lira¹, Carlos Molina², and Antonio González³

¹ Universitat Politècnica de Catalunya

² Universitat Rovira i Virgili

³ Intel Barcelona Research Center, Intel Labs - UPC

Abstract. In response to the constant increase in wire delays, Non-Uniform Cache Architecture (NUCA) has been introduced as an effective memory model for dealing with growing memory latencies. This architecture divides a large memory cache into smaller banks that can be accessed independently. Banks close to the cache controller therefore have a faster response time than banks located farther away from it. In this paper, we propose and analyse the insertion of an additional bank into the NUCA cache. This is called *Last Bank*. This extra bank deals with data blocks that have been evicted from the other banks in the NUCA cache. Furthermore, we analyse the behaviour of the cache line replacements done in the NUCA cache and propose two optimisations of Last Bank that provide significant performance benefits without incurring unaffordable implementation costs.

1 Introduction

Advances in technology have made it possible for more and more transistors to be placed on a single chip. This enables computer architects to deal with a huge number of transistors and build smarter processors. In the 1990s, the main trend in constructing computers to satisfy the high performance requirements of different applications was to increase clock speed and introduce even greater complexity in order to exploit Instruction-Level Parallelism (ILP) in applications. While increasing clock speed improves the performance of the processors, it also increases power consumption. At the end of the 1990s, power efficiency became a critical issue because the power consumption per unit of area was growing dramatically and almost reaching the limits of affordability. Another disadvantage of increasing clock speed is that, as memory frequency does not grow as fast as processor frequency, the gap between processor speed and memory speed has widened dramatically.

Today it is feasible to integrate more than one processor on a single chip. This has opened up a new era in computer architecture (called the *Multicore era*) in which the smarter integration of more than one core on a single chip becomes the new challenge [1]. Chip Multiprocessors (CMPs) can work at much lower clock frequencies than single core processors, thus solving the power consumption

problem. CMPs also moderate the effects of the broad gap between processor and memory speed by exploiting Thread-Level Parallelism (TLP). However, new issues arise from this novel architecture. Chip complexity has increased with the introduction of multiple processors. A high performance on-chip interconnection network is required to connect all the components on the die. One of the main advantages of CMPs over other multiprocessors is the on-chip network, which enables data to be shared at very low latency.

The memory subsystem is also more complex in CMPs. Access to the main memory is very expensive due to main memory response time and network delay. Having a shared last-level cache on-chip is therefore a key issue in this kind of architecture. Today's CMP architectures incorporate larger and more complex cache hierarchies. Recent studies have proposed mechanisms for dealing with new challenges to the memory system posed by CMP architectures, some of the most notable of these being cooperative caching [23], victim replication [4], adaptive selective replication [5] and other works that exploit the private/shared cache partitioning scheme [67].

The increasing influence of wire delay in cache design means that access latencies to the last-level cache banks are no longer constant [8]. Non-Uniform Cache Architectures (NUCAs) have been proposed [9] as a way of addressing this problem. NUCA divides the whole cache memory into smaller banks and allows nearer cache banks to have lower access latencies than farther banks, thus mitigating the effects of the internal wires of the cache.

In this paper we propose and analyse the insertion of an additional bank into the NUCA cache. This bank, called *Last Bank*, acts as a last-level cache on the chip by dealing with data blocks evicted from the NUCA cache. Moreover, we propose two optimisations for the Last Bank mechanism: *Selective Last Bank* and *LRU prioritising Last Bank*.

The remainder of this paper is structured as follows. Section 2 describes the baseline architecture assumed in our studies. Experimental methodology is presented in Section 3. Section 4 introduces the Last Bank mechanism. Section 5 analyses the NUCA cache line replacements and motivates the Last Bank optimisations proposed in Section 6. Our results are analysed in Section 7. Related work is discussed in Section 8, and concluding remarks are given in Section 9.

2 Baseline Model

As illustrated in Figure 1, the baseline architecture consists of an eight-processor CMP based on that of Beckmann and Wood [10]. The processors are located on the edges of the NUCA cache, which occupies the central part of the chip. Each processor provides the first-level cache memory, composed of two separated caches: one for instructions and one for data. The NUCA cache is then the second-level cache memory and it is shared by the eight processors. The NUCA cache is divided into 256 banks structured in a 16x16 mesh that are connected via a 2D mesh interconnection network. The banks in the NUCA cache are also

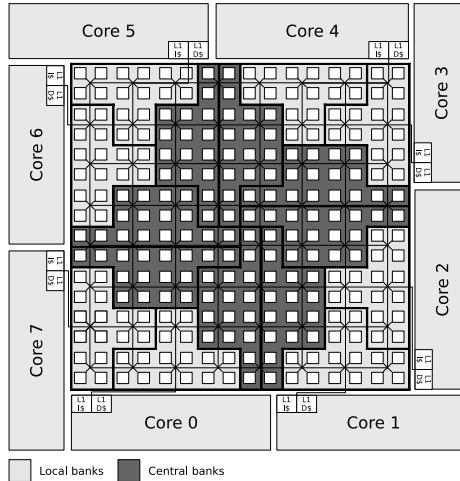


Fig. 1. Baseline architecture layout

logically separated into 16 banksets that are either *local banks* (lightly shaded in Figure 1) or *central banks* (darkly shaded in Figure 1) in accordance with their physical distance to the processors.

3 Experimental Framework

We used the full-system execution-driven simulator, Simics [11], extended with the GEMS toolset [12]. GEMS provides a detailed memory-system timing model that enabled us to model the NUCA cache architecture. The simulated architecture is structured as a single CMP made up of eight homogeneous cores. Each core is a superscalar out-of-order SPARCv9 processor modelled by the Opal simulator, which is one of the extensions that GEMS provides for Simics. With regard to memory hierarchy, each core provides a first-level cache, which is in fact two caches: one for instructions and one for data. The second level of the memory hierarchy is the NUCA cache. In order to maintain correctness and robustness in the memory system we used the MOESI token-based coherence protocol. Table 1 summarises the configuration parameters assumed in our studies. The access latencies of the memory components are based on models done with the CACTI 6.0 [13] modelling tool, which is the first version of CACTI that provides support for modelling NUCA caches.

We simulated a set of workloads from the PARSEC [14] benchmarks with *simlarge* input data sets. The method we used for the simulations consisted of, first skipping both the initialisation and thread creation phases, then fast-forwarding while warming all caches for 100 million instruction intervals, and finally performing a detailed simulation for 200 million instruction intervals.

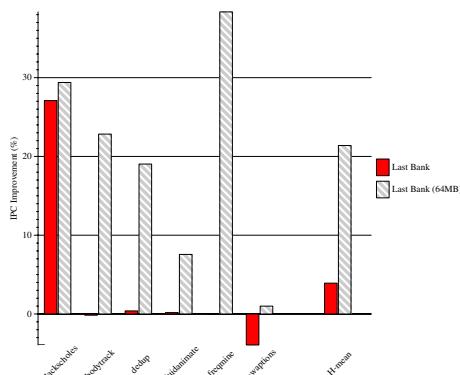
Table 1. Configuration parameters

Processors	8, 4-way SMT
Branch Predictor	YAGS
Instr. Window / ROB	64 / 128 entries
Block size	64 bytes
L1 Cache (Instr./Data)	8 KBytes, 4-way / 8 KBytes, 4-way
L2 Cache (NUCA)	1 MByte, 4-way, 256 Banks
NUCA Bank	4 KBytes, 4-way
L1 Latency	3 cycles
NUCA Bank Latency	2 cycles
Router Latency	1 cycle
Memory Latency	350 cycles (from core)

4 Last Bank

Cache memories take advantage of the temporal and spatial data locality that applications usually exhibit. However, the whole working set does not usually fit into the cache memory, causing capacity and conflict misses. These misses mean that a line that may be accessed later has to leave the cache prematurely. As a result, evicted lines that are later reused return to the cache memory in a short period of time. This is more pronounced in a NUCA cache memory because data movements within the cache are allowed, so the most recently accessed data is concentrated in a few banks rather than spread over the entire cache memory. Therefore, we propose adding an extra bank to deal with data blocks that have been evicted from the NUCA cache. This extra bank, called *Last Bank*, provides evicted data blocks a second chance to come back to the NUCA cache without leaving the chip.

Last Bank, which is as large as a single bank in the NUCA cache, acts as the last-level cache between the NUCA cache and the off-chip memory. It is physically located in the centre of the chip at about the same distance to all

**Fig. 2.** Speed-up achieved with Last Bank compared to baseline configuration

cores. When there is a hit on Last Bank, the data block that is being accessed leaves Last Bank and goes back to the corresponding bank in the NUCA cache.

Figure 2 shows the performance gain achieved by adding Last Bank to the baseline configuration. On average, the Last Bank configuration outperforms the baseline with an IPC improvement of 4%. On the other hand, we also evaluated an unaffordable setup in which the evicted data blocks nearly always fit in Last Bank. The 64-MByte-sized Last Bank configuration shows the significant performance potential of this proposal, that is, an average IPC improvement of 22%. Thus, we conclude that although adding Last Bank to the baseline configuration results in performance benefits, these benefits are strictly limited by the size of Last Bank. The following sections analyse the behaviour of the evicted lines and propose two optimisations that provide extra IPC improvement to the Last Bank proposal without incurring unaffordable implementation costs.

5 Characterization of NUCA Cache Line Replacements

This section analyses the behaviour of the data blocks that leave the NUCA cache, focusing on those that later return. The aim of this study is to find hints to help us guess whether a data block that is being evicted from the NUCA cache will be accessed later.

5.1 Frequency of Insertions into NUCA Cache of Reused Addresses

We first analysed the frequency of insertions of reused addresses to find out whether most reinsertions are concentrated on a few addresses or whether the number of reinsertions are fairly spread out over all the reused addresses.

Figure 3(a) shows the insertion frequency of addresses that have been previously evicted for each of the workloads simulated (note that the X-axis deals with an exponential scale). The figure shows that 16% of reused addresses represent almost 50% of total reinsertions. This indicates that the assumed workloads generally concentrate a huge number of reinsertions on a few addresses. In this case, if we prevent this 16% of addresses from being evicted with a selective mechanism, off-chip requests would reduce dramatically as almost 50% of total reinsertions would become on-chip requests. Performance would therefore improve accordingly.

5.2 Time between Eviction and Insertion of a Reused Address

The second part of the analysis focused on the cycles passed between the eviction of a reused address and the reinsertion of the same address.

Figure 3(b) presents, for each workload, the percentage of addresses that return to the NUCA cache in a certain period of cycles. On average, nearly 30% of evicted addresses that are later inserted into the NUCA cache return in less than 100,000 cycles. In fact, with the *blackscholes* workload over 50% of evicted addresses return in less than 1,000 cycles. This clearly explains the benefit obtained by the Last Bank configuration.

5.3 Last Location of Evicted Data Blocks

This section analyses the location of evicted data blocks before replacement in order to evaluate whether the probability of an evicted data block being reinserted into the NUCA cache is related to the last NUCA bank in which it was stored. For the sake of simplicity, we have classified the NUCA banks according to the two bank categories introduced in Section 2 (*local* and *central*).

The results shown in Figure 3(c) are classified into the following three categories: 1) evicted data blocks that are not later accessed (0 R), 2) evicted data blocks that are later accessed only once more (1 R), and 3) evicted data blocks that are later accessed more than once (+1 R).

Figure 3(c) shows that, on average, the vast majority of addresses that were evicted from a local bank were later reinserted once or more than once (over 60% and over 70% of reused addresses, respectively). Moreover, fewer than 60% of the addresses that were not accessed after eviction were evicted from a local bank. In general, this trend in which the percentage of the non-reinserted evicted addresses from a local bank is lower than that of the evicted and later inserted addresses is consistent for all workloads.

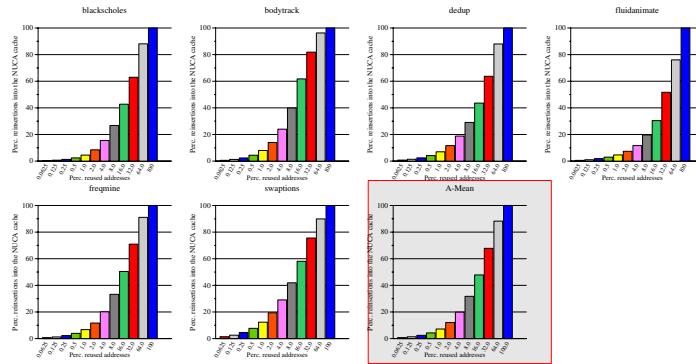
5.4 Action That Provokes Replacement

Finally, we analysed the type of action that provokes the eviction of the data block from the NUCA cache. Only two actions motivate an eviction from a NUCA cache bank: *incoming data from memory (New Data)* or *an eviction from L1 cache (L1 replacement)*. When an incoming data block from off-chip memory is inserted into the NUCA cache, it is placed statically in any of the banks of the entire NUCA architecture. On the other hand, when a data block comes from an L1 cache eviction, it is always placed in the closest local bank to the L1 cache that provoked the replacement.

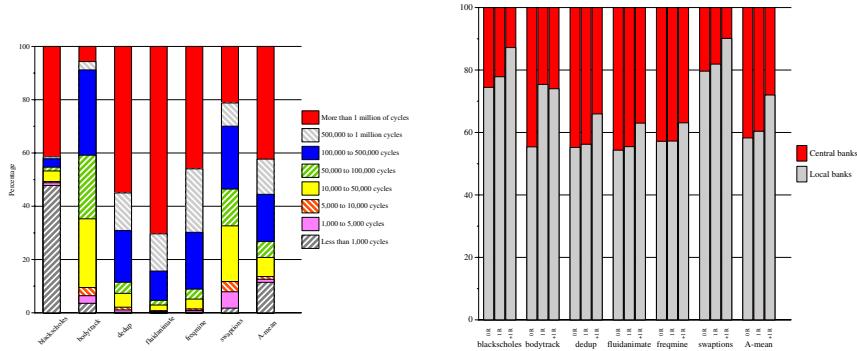
Figure 3(d) shows the percentage of evictions provoked by each of the actions, broken down into the following categories: addresses that were not accessed after their eviction; addresses that were inserted into the NUCA cache only once after their eviction; and addresses that were evicted and later inserted into the NUCA cache more than once. On average, 55% of the evictions of addresses that were reused more than once were evicted due to L1 replacement. Meanwhile, the same action provoked nearly 45% of the evictions of addresses that were reinserted only once into the NUCA cache. On average, only 40% of non-reused addresses were evicted due to an L1 cache eviction. In general, for all workloads the percentage of evictions provoked by L1 cache replacements that are not accessed later is always lower than in the other two cases.

6 Last Bank Optimisations

Section 4 shows that some performance benefits are achieved by adding Last Bank to the baseline NUCA cache architecture. However, it also shows

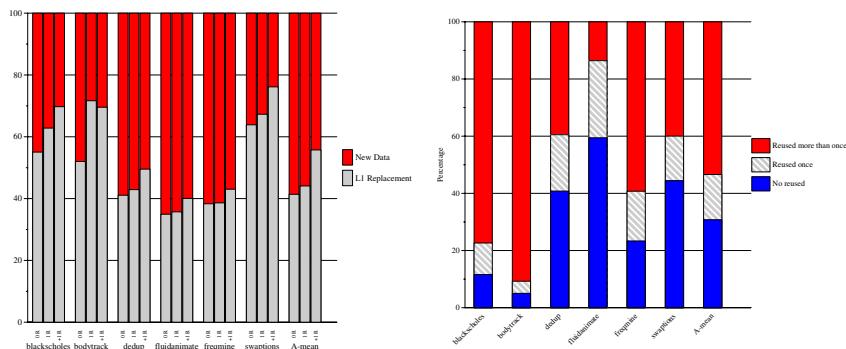


(a) Frequency of insertions into NUCA cache of addresses previously evicted.



(b) Cycles between eviction and insertion of the same address.

(c) Percentage of reused addresses per bank type (local or central).



(d) Action that provokes eviction.

(e) Percentage of reused addresses in the NUCA cache.

Fig. 3. Analysis of NUCA cache line replacements

considerable potential improvements in performance that cannot be attained using the current Last Bank configuration. This section introduces two mechanisms that optimise the usage of Last Bank based on the analysis described in Section 5.

6.1 Selective Last Bank

Section 5.2 shows that almost 30% of evicted addresses that are later accessed return to the NUCA cache in a reduced number of cycles (fewer than 100,000 cycles). However, Last Bank cannot take advantage of this fast return and, with the exception of *blackscholes*, it does not result in any IPC improvement at all (see Figure 2). This is mainly because Last Bank is not large enough to deal with all the evicted data blocks from the entire NUCA cache. So, Last Bank is polluted with *useless* data blocks that will not be accessed again and that provoke the eviction of *useful* data blocks from Last Bank before they are accessed.

Based on these observations, we propose a selection mechanism in Last Bank called *Selective Last Bank*. This selection mechanism allows evicted data blocks to be inserted into Last Bank by way of a filter. The aim is to prevent Last Bank from becoming polluted with data blocks that are not going to be accessed further by the program.

Section 5.3 shows that almost 70% of reused addresses were evicted from local banks. On the other hand, only 60% of addresses that were not requested further after NUCA cache eviction were evicted from local banks and the rest were evicted from central banks. We therefore propose a filter that allows only the evicted data blocks that resided in a local bank before eviction to be cached.

Finally, the action that provokes the eviction from the NUCA cache could provide another filter for *Selective Last Bank*. However, after evaluating the filters for both incoming data from the off-chip and for L1 cache evictions, they were both ruled out because they complement one another and do not provide benefits to Last Bank.

6.2 LRU Prioritising Last Bank

Figure 3(e) shows that the vast majority of evicted lines that return to the cache memory leave the NUCA cache and are later reused at least twice. Thus, we propose modifying the data eviction algorithm of the NUCA cache in order to prioritise the lines that enter the NUCA cache from Last Bank. We call this *LRU prioritising Last Bank (LRU-LB)*. LRU-LB gives the lines that have been stored by the Last Bank and that return to the NUCA cache an extra chance, so they remain in the on-chip cache memory longer. This requires storing an extra bit, called the *priority bit*, attached to each line in the NUCA cache.

The LRU-LB eviction policy works as follows. When an incoming line comes to the NUCA cache memory from Last Bank, its priority bit is set. Figure 4(a) shows how this policy works when a line with its priority bit set is in the LRU position. The line that currently occupies the LRU position, then, clears its priority bit and updates its position to the MRU, and thus, the other lines in the

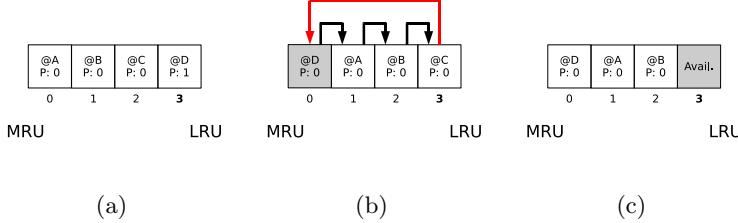


Fig. 4. LRU prioritising Last Bank (LRU-LB) scheme. (a) A priority line is in the LRU position. (b) The priority line resets its priority bit and updates its position to the MRU; the other lines move one position forward to the LRU. (c) The line in the LRU position is evicted since its priority is clear.

LRU stack move one position forward towards the LRU (Figure 4(b)). Finally, as the line that is currently in the LRU position has its priority bit clear, it is evicted from the NUCA cache (Figure 4(c)). If the line that ends in the LRU position has its priority bit set, the algorithm described above is applied again until the line in the LRU position has its priority bit clear.

7 Results and Analysis

This section analyses the performance results obtained with the two optimisations for the Last Bank proposed in Section 6, *Selective Last Bank* and *LRU prioritising Last Bank (LRU-LB)*. With Selective Last Bank, the filter only allows blocks that have been evicted from a local bank to be cached.

Figure 5 shows that, on average, both Last Bank optimisations achieve greater IPC improvement than that achieved by the Last Bank configuration. Selective Last Bank obtains 6% speed-up with respect to the baseline configuration, while the performance gain obtained by LRU-LB is nearly 8%.

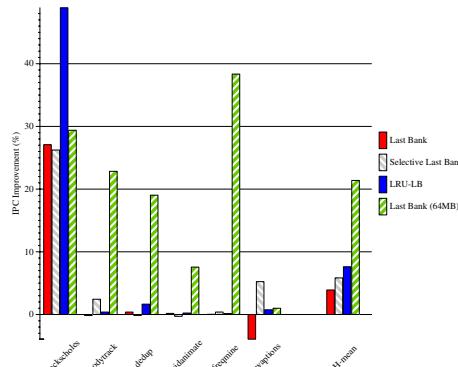


Fig. 5. Speed-up achieved with Last Bank optimisations

Figure 5 also shows that Selective Last Bank does not obtain performance benefits in three of the workloads (*blackscholes*, *dedup* and *fluidanimate*), however, it works especially well with *bodytrack* and *swaptions*. It is clear, then, that with an effective filter Selective Last Bank would provide significant performance benefits. Adaptive selective filters for Selective Last Bank will be analysed in future work.

With regard to the LRU-LB optimisation, giving an extra chance to reused addresses before being evicted from the NUCA cache has two direct consequences: 1) if accessed, they are closer to cores, and due to the NUCA basis they have lower access latency, and 2) the number of reused addresses stored in the NUCA cache is higher. As a result, LRU-LB outperforms the Last Bank configuration with all simulated workloads. We highlight that with the *blackscholes*, LRU-LB dramatically outperforms the Last Bank configuration (even the unaffordable setup) by achieving 49% IPC improvement.

8 Related Work

Kim et al. [9] introduced the concept of Non-Uniform Cache Architecture (NUCA). They observed that the increase in wire delays would make cache access times no longer a constant. Instead, latency would become a linear-function of the line's physical location within the cache. From this observation, several NUCA architectures were designed by partitioning the cache into multiple banks and using a switched network to connect these banks. The two main architectures, however, were Static NUCA (S-NUCA) and Dynamic NUCA (D-NUCA). Both designs organise the multiple banks into a two-dimensional switched network. The difference between the two architectures is the *Placement Policy* they manage. While in S-NUCA architecture, data are statically placed in one of the banks and always in the same bank, in D-NUCA architecture data can be promoted to be placed in closer and faster banks. Since the development of these two architectures, several works using NUCA architectures have appeared in the literature. One of the most relevant proposals is NuRAPID [15], which decouples data and tag placement. NuRAPID stores tags in a bank close to the processor, optimising tag searches. Whereas NUCA searches tag and data in parallel, NuRAPID searches them sequentially. This increases overall access time but provides greater power efficiency. Another difference between NUCA and NuRAPID is that NuRAPID partitions the cache into fewer, larger and slower banks. In terms of performance, NuRAPID and D-NUCA achieve similar results, but NuRAPID vastly outperforms D-NUCA in power efficiency.

However, the introduction of CMP architectures posed additional challenges to the NUCA architecture leading Beckmann and Wood [10] to analyse NUCA for CMP. They demonstrated that block migration is less effective for CMP because 40-60% of the hits in commercial workloads were satisfied in the central banks. Block migration effectively reduced wire delays in uniprocessor caches. However, to improve CMP performance, the capability of block migration relied on a smart search mechanism that was difficult to implement. Chishti et al. [15]

also proposed a version of NuRAPID for CMP in which each core had a private tag array instead of a single, shared tag array.

Recent studies have explored policies for bank placement [16], bank migration [17] and bank access [18] in NUCA caches. Kim et al. [9] proposed two alternatives for bank replacement policy: *zero-copy policy* and *one-copy policy*. *Zero-copy policy* means that an evicted data element is sent back to the off-chip memory. In *one-copy policy*, on the other hand, the victim data element is moved to a lower-priority bank farther from the processor.

9 Conclusions

In this paper we propose adding an extra bank to deal with data blocks evicted from the NUCA cache. We call this bank *Last Bank*. This extra bank, which acts as the last-level cache on the chip, gives evicted data blocks a second chance to be reinserted into the NUCA cache without leaving the chip. We found that although this mechanism provides significant performance potential, the benefits achieved with Last Bank are strictly limited by the number of lines that can be allocated. Therefore, we analysed the behaviour of the cache line replacements in the NUCA cache and propose two optimisations that provide additional IPC improvement to Last Bank without incurring unaffordable implementation costs.

We first propose *Selective Last Bank*, which defines that only evicted data blocks that pass the selection filter are allowed to be allocated in Last Bank. In this paper we propose a selection filter that allows only data blocks that were evicted from a local bank to be cached. With Selective Last Bank we achieve up to 6% IPC improvement with respect to the baseline configuration, however, the performance results obtained by this mechanism rely on the effectiveness of the filter applied. In future works we will analyse adaptive selection filters.

Finally, we propose *LRU prioritising Last Bank*. This modifies the data eviction algorithm of the NUCA cache by prioritising the data blocks that came from Last Bank. On average this mechanism achieved performance gains of up to 8% with respect to the baseline configuration.

Acknowledgements

This work is supported by Spanish Ministry of Science and Innovation (MCI) and FEDER funds of the EU under contracts TIN 2007-61763 and TIN 2007-68050-C03-03, Generalitat de Catalunya under grant 2005SGR00950, and Intel Corporation. Javier Lira is funded by MCI-FPI grant BES-2008-003177.

References

1. Gorder, P.F.: Multicore processors for science and engineering. In: Computing in Science & Engineering (March-April 2007)
2. Chang, J., Sohi, G.S.: Cooperative caching for chip multiprocessors. In: Proc. of the 33rd International Symposium on Computer Architecture, ISCA 2006 (2006)

3. Chang, J., Sohi, G.S.: Cooperative cache partitioning for chip multiprocessors. In: Procs. of the 21st ACM International Conference on Supercomputing, ICS-21 (2007)
4. Zhang, M., Asanović, K.: Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In: Procs. of the 32nd International Symposium on Computer Architecture, ISCA 2005 (2005)
5. Beckmann, B.M., Marty, M.R., Wood, D.A.: Asr: Adaptive selective replication for cmp caches. In: 39th Annual IEEE/ACM International Symposium of Microarchitecture, MICRO-39 (2006)
6. Dybdahl, H., Stenström, P.: An adaptive shared/private nuca cache partitioning scheme for chip multiprocessors. In: IEEE 13th International Symposium on High-Performance Computer Architecture (2007)
7. Guz, Z., Keidar, I., Kolodny, A., Weiser, U.C.: Nahalal: Cache organization for chip multiprocessors. IEEE Computer Architecture Letters (2007)
8. Agarwal, V., Hrishikesh, M.S., Keckler, S.W., Burger, D.: Clock rate vs. ipc: The end of the road for conventional microprocessors. In: 27th International Symposium on Computer Architecture (2000)
9. Kim, C., Burger, D., Keckler, S.W.: An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In: 10th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS (October 2002)
10. Beckmann, B.M., Wood, D.A.: Managing wire delay in large chip-multiprocessor caches. In: 37th International Symposium on Microarchitecture, MICRO-37 (2004)
11. Magnusson, P.S., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Höglberg, J., Larsson, F., Moestedt, A., Werner, B.: Simics: A Full System Simulator Platform. Computer 35(2), 50–58 (2002)
12. Martin, M.M.K., Sorin, D.J., Beckmann, B.M., Marty, M.R., Xu, M., Alameldeen, A.R., Moore, K.E., Hill, M.D., Wood, D.A.: Multifacet's general execution-driven multiprocessor simulator (gems) toolset. Computer Architecture News (September 2005)
13. Muralimanohar, N., Balasubramonian, R., Jouppi, N.P.: Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0. In: 40th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-40 (2007)
14. Bienia, C., Kumar, S., Singh, J.P., Li, K.: The parsec benchmark suite: Characterization and architectural implications. In: Procs. of the 17th International Conference on Parallel Architectures and Compilation Techniques (October 2008)
15. Chishti, Z., Powell, M.D., Vijaykumar, T.N.: Distance associativity for high-performance energy-efficient non-uniform cache architectures. In: Proceedings of the 36th International Symposium on Microarchitecture, MICRO-36 (2003)
16. Huh, J., Kim, C., Shafi, H., Zhang, L., Burger, D., Keckler, S.W.: A nuca substrate for flexible cmp cache sharing. In: Procs. of the 19th ACM International Conference on Supercomputing, ICS-19 (2005)
17. Kandemir, M., Li, F., Irwin, M.J., Son, S.W.: A novel migration-based nuca design for chip multiprocessors. In: ACM/IEEE conference on Supercomputing (2008)
18. Muralimanohar, N., Balasubramonian, R.: Interconnect design considerations for large nuca caches. In: Procs. of the 34th International Symposium on Computer Architecture, ISCA 2007 (2007)

Paired ROBs: A Cost-Effective Reorder Buffer Sharing Strategy for SMT Processors

R. Ubal, J. Sahuquillo, S. Petit, and P. López

Department of Computing Engineering (DISCA)

Universidad Politécnica de Valencia, Valencia, Spain

raurte@gap.upv.es, {jsahuqui,spetit,plopez}@disca.upv.es

Abstract. An important design issue of SMT processors is to find proper sharing strategies of resources among threads. This paper proposes a ROB sharing strategy, called *paired ROB*, that considers the fact that task parallelism is not always available to fully utilize resources of multithreaded processors. To this aim, an evaluation methodology is proposed and used for the experiments, which analyzes performance under different degrees of parallelism. Results show that paired ROBs are a cost-effective strategy that provides better performance than private ROBs for low task parallelism, whereas it incurs slight performance losses for high task parallelism.

1 Introduction

As a single software task is far from exploiting peak performance of current superscalar processors, simultaneous multithreading (SMT) [1] was proposed as a way of increasing the utilization of the processor functional units. One of the main research challenges of SMT processors is the design and optimization of proper resource allocation policies that decide how processor resources are assigned to each thread. Related proposals focus on the distribution of bandwidth resources, such as fetch or issue width [2][3], as well as storage resources, such as instruction queue, load-store queue or physical register file [4][5][6]. Most storage resources can be easily distributed in a dynamic way among threads, since any of their free entries can be allocated/deallocated at any time. However, the reorder buffer (ROB) manages instructions from threads in a FIFO order, which lowers the flexibility of dynamically assigning ROB entries to different threads.

There are two basic strategies to share the ROB among threads. The first one is referred in this paper to as *private ROB*, and consists in statically partitioning the ROB among threads. This approach is simple, has small hardware cost, and distributes ROB resources fairly among threads. The second strategy is referred to as *one-queue ROB*, and consists in a shared queue where instructions from different threads are inserted in local FIFO order, but in any global order. This approach has, depending on the implementation, several drawbacks in terms of hardware cost and performance: a) the dispatch, commit, and recover logic increases, b) instructions from different threads can block each other at commit, c) ROB *holes* (empty intermingled slots) appear when one single thread squashes

its instructions after a mispeculation, and d) global performance can be reduced when a stalled thread occupies too many entries of the shared ROB.

All these arguments against the shared approach seem enough to discard this design. This conclusion has been already drawn in several research works [4][7][8]. However, all these studies evaluate ROB partition strategies in heavily loaded systems, that is, systems running a number of computation intensive software tasks equals to the number of hardware threads (e.g., a 4-benchmark mix on a 4-threaded 8-way processor). But it is not uncommon to find lightly loaded multithreaded systems, that is, systems using only a small portion of their hardware threads for computation intensive workloads.

Let us consider a typical user executing a single sequential (i.e., non parallel) computation intensive application on a 2-threaded SMT processor. This user does not experience any performance improvement at all with multithreading. Moreover, if the processor implements a private ROB partitioning, only half of the ROB entries are available to the computation intensive application, so a performance loss can be noticed for the same total ROB size. This effect has been evaluated by running the SPEC2000 benchmark suite on a single-threaded and a 2-threaded processor, both of them with the baseline parameters shown in Section 4, and a total number of 128 ROB entries. Results (not shown here) point out a performance loss of 9.2% and 20.2% for integer and floating-point benchmarks, respectively, when using the multithreaded machine with private ROBs.

Of course, multithreaded processors should not be evaluated only in lightly loaded systems, which would be unfair. On the other hand, to assume that the system is always running a heavy workload is neither a realistic way to measure performance. We claim that this is an issue that needs to be considered for a fair evaluation of multithreaded processors.

In this work, we propose a ROB sharing strategy, called *paired ROB*, that takes into consideration the fact that task parallelism is not always available to fully utilize resources of multithreaded processors. First, an evaluation methodology is proposed based on a formal definition of the available task parallelism. Then, paired ROBs are shown to perform similarly to shared ROBs for low task parallelism (with a lower hardware cost), and close to private ROBs for high task parallelism.

The rest of this paper is structured as follows. Section 2 analyzes advantages and disadvantages of private and shared ROBs, and describes the paired ROB proposal. Section 3 defines in a formal way the available workload parallelism, and details the simulation methodology. Section 4 shows simulation results, and finally, some related work and concluding remarks are presented.

2 ROB Sharing Strategies

In this section, three different ROB sharing strategies are analyzed, namely private, one-queue (shared) and paired ROBs. The first two sharing strategies represent basic design options, which suffer from some disadvantages regarding either complexity or performance. These drawbacks are smoothed by paired ROBs, by trying to gather the best properties of both approaches.

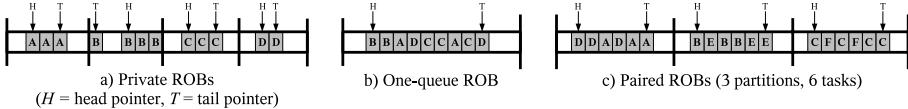


Fig. 1. ROB sharing strategies

The hardware cost analysis presented in this section is based on the required accesses to the ROB, which usually occur in the *dispatch* (insertion), *commit* (extraction), and *recover* (squash) stages of the processor pipeline. In what follows, the number of threads supported by a multithreaded system is represented as n , while s stands for the *total* ROB size in number of entries (i.e., the sum of all per-thread ROB sizes). For the sake of simplicity, a one-way processor is assumed, but results can be easily generalized to w -way superscalar processors, where ROBs are implemented by using w -port RAMs [9].

2.1 Private ROBs

The first straightforward sharing strategy consists in not sharing the ROB at all. The global ROB is split into different partitions, which act as independent FIFO queues, each one associated to a thread (Figure 1a). Each queue is managed by two pointers, namely *head* (H) and *tail* (T), which track the insertion and extraction point into and from the corresponding queue. The required hardware structures and the actions involving the ROB are the following:

At the dispatch stage, instructions are placed in the ROB at the position indicated by the tail pointer. To this aim, a demultiplexer (or *decoder*), controlled by the ROB tail pointer, sends the instruction information to the corresponding ROB entry. Private ROBs require n independent demultiplexers with $\frac{s}{n}$ outputs, and an additional n -output demultiplexer selects the target private ROB.

At the commit stage, instructions are extracted from the ROB head. Private ROBs can perform this step in parallel with $n \frac{s}{n}$ -input multiplexers, which transmit the extracted instruction into the rest of the commit logic.

Finally, the mispeculation recovery mechanism squashes some or all instructions in the ROB. If recovery is performed at the commit stage, the ROB must be just emptied, that is, the head and tail pointers must be reset. On the contrary, if recovery is implemented at the writeback stage, only instructions younger than the recovered one must be squashed, which implies an adjustment of the tail pointer. In any case, recovery requires no read, write, or selection of ROB entries; related hardware must just update the queue pointers.

2.2 One-Queue ROB

In the opposite extreme, the *one-queue ROB* is based on a fully shared circular queue, where all threads place their instructions in the global dispatch order (Figure 1b). There are some advantages and drawbacks in this approach, quantitatively evaluated in Section 4, and described next:

Disadvantages

Inter-thread blocking at commit. The fact that two or more threads share a single FIFO queue can lead to a situation in which the oldest instruction of a thread is ready to commit, but unable to do it for not being located at the ROB head. If the ROB head is occupied by an uncompleted instruction of any other thread, a waste of commit bandwidth is incurred.

Holes at recovery. In the one-queue ROB, instructions from different threads can be intermingled along the structure. This fact breaks the simplicity of a recovery mechanism in which only pointer updates are performed. Instead, only those instructions corresponding to the recovered thread are squashed on mispeculation, leaving empty intermingled slots or holes. These holes remain in the ROB until their are drained at the commit stage, and cannot be assigned to new decoded instruction, so they can cause a premature pipeline stall due to lack of space in the ROB.

Thread starvation. An intuitively potential advantage of a one-queue ROB is that instructions of any hardware thread may occupy all s ROB entries, in contrast to private ROBs, which restrict the ROB usage of a single thread to $\frac{s}{n}$ entries. Although this property brings flexibility, it has been demonstrated that benefits are not straightforwardly achieved [4]. In fact, performance losses occur when stalled threads (e.g., due to a long memory operation) uselessly occupy large portions of the shared ROB, preventing active ones from inserting new instructions into the pipeline.

Regarding hardware cost, one-queue ROBs have a more expensive implementation than private ROBs. For the same total number of entries s , the complexity of hardware structures increases. For dispatch, an s -output demultiplexer is needed, which has a higher delay and area than the $n \frac{s}{n}$ -output demultiplexers of the private approach; for commit, an s -input multiplexer is required; finally, recovery can be implemented with a bit line per thread that turns the associated instructions into empty slots.

Advantages

Though the cited disadvantages could make one believe that shared ROBs do not deserve any interest at all, there are situations where this approach completely outperforms any other sharing strategy. Consider a multithreaded processor supporting 8 hardware threads and using private ROBs. A highly intensive computation environment can take full advantage of multithreading in such machine by running 8 threads on it 100% of the time. However, 8 tasks may not be always available in the system.

The opposite extreme (discarding utter idleness of the system) is the case where only one single software context is being executed. Private ROBs constrain to use only $\frac{1}{8}$ of the total number of ROB entries available. However, a shared ROB permits a full occupation of the ROB by the single software context, enabling 8 times more instructions in flight, while not incurring any previously cited performance-related disadvantage of the one-queue ROB.

2.3 Paired ROBs

After analyzing the pros and cons of private and one-queue ROBs, a new sharing strategy, referred to as *paired ROB*, is proposed with the aim of gathering the main advantages of both private and one-queue ROBs.

Assuming a multithreaded processor with n hardware threads, and being s the total number of available ROB entries, the global ROB is partitioned into $\frac{n}{2}$ independent structures (see Figure 11c). Each partition can be occupied by instructions from at most two threads. The limit of two threads per partition is supported by the results discussed in Section 4.1.

If the system executes less tasks than available hardware threads, the assignment of ROB partitions to tasks is carried out in such a way that the number of tasks sharing a ROB partition is minimized. In other words, new active threads try to allocate an empty ROB partition before starting to share other occupied partition. This policy avoids inter-thread blocking, holes at recovery and thread starvation as long as it is possible.

The benefits of paired ROBs lie both in terms of hardware complexity and performance. Regarding hardware complexity, paired ROBs need $\frac{n}{2} \frac{2s}{n}$ -output demultiplexers for dispatch, $\frac{n}{2} \frac{2s}{n}$ -input multiplexers for commit, and a single bit line for recovery (only instructions from two different threads are queued in each ROB). The number of queue pointer sets is also reduced to $\frac{n}{2}$. This complexity is higher than for private ROBs, but lower than for one-queue ROBs.

3 Multitask Degree (MTD)

The effectiveness of a specific ROB sharing strategy can strongly vary depending on the number of tasks running on the system. To take this effect into account, the concept of Multitask Degree (*MTD*) is first defined in this section.

We define the *MTD* as a value between 0 and 1 that represents the average task parallelism present in a non-idle multithreaded system. A system with a value of $MTD = 0$ is characterized by running a single task all the time, while a value of 1 means that the system is fully loaded, that is, the number of tasks matches the number of hardware threads during all the execution time. Analytically, the *MTD* can be expressed as $\sum_{i=1}^n \frac{i-1}{n-1} t_i$, where n is the number of hardware threads of the system, and t_i is the fraction of the total execution time in which there are exactly i tasks running in the system. For example, in a 2-threaded processor, a value of $MTD = 0.5$ means that 50% of the time only one task is running, while two tasks are running the rest of the time. In general, the *MTD* value establishes the average number of running tasks as $MTD*(n-1)+1$. Figure 2 plots this equation for a 2- and 4-threaded system.

For the specific case of 2-threaded systems, the *MTD* conveys univocal information about the distribution of the system load. For example, a 2-threaded system with $MTD = 0.3$ is characterized by executing 1 task 30% of the time and 2 tasks 70% of the time. The *MTD*, however, does not provide any information about the load distribution of a system with more than 2 threads. For

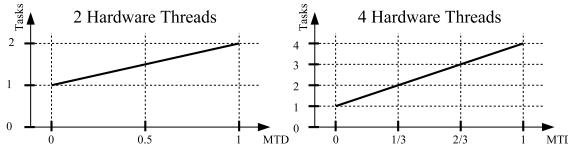


Fig. 2. Number of tasks as a function of the MTD value in 2- and 4-threaded processors

example, a 5-threaded processor with an MTD of 0.5 (which has an average load of $0.5 * (5 - 1) + 1 = 3$ tasks) can either execute 50% of the time 1 task and 5 tasks the rest of the time, or it can execute 2 and 4 tasks during equal portions of the time, or it can execute 3 tasks all the time, etc. In other words, there are multiple combinations of system loads and fractions of time that lead to the same resulting average value.

In order to cut down the number of possible combinations and simplify the analysis, we define the concept of *steady multithreaded processor*. An n -threaded processor is said to be *steady*, or to have a *steady load*, when the number of running tasks is exactly either x or $x + 1$ ($1 \leq x < n$). In this way, we limit the model to execution periods where the number of active threads can increase or decrease at most by one.

With this simplification, the number of possible load distributions is reduced to just one, which means that the MTD indeed characterizes the system load distribution univocally in a steady multithreaded processor. For instance, a steady 3-threaded system with $MTD = 0.5$ can be only executing 2 tasks all the time, while the same system with $MTD = 0.25$ can be only achieved by executing 1 task 50% of the time and 2 tasks the rest of the time. Finally, notice that this simplification keeps the model realistic, since operating systems usually assign processes or threads to the CPU in an incremental way. When applied to steady machines, the MTD is hereafter referred to as *Steady Multitask Degree (sMTD)*.

3.1 Simulation Methodology

Experimental results presented in this paper evaluate different ROB sharing strategies on multithreaded processors with 2, 4, and 8 hardware threads. For each of them, different levels of task parallelism are evaluated, by presenting results as a function of $sMTD$. The algorithm used to obtain a continuous range of performance results from a finite set of simulations is presented next. Consider a steady multithreaded system with n hardware threads, in which a specific sharing strategy is evaluated.

First, n groups of simulations are launched. In the first group, only one task is run on the system, and each simulation uses a different benchmark from the SPEC2000 suite. In the second group, each experiment executes two instances of the same benchmark, each of them running on a different hardware thread, and so on.

Each simulation provides a performance value (IPC), whose harmonic mean is computed independently for each group. In this way, n average performance values are obtained, namely $IPC_1, IPC_2, \dots, IPC_n$. IPC_1 stands for the average performance of the n -threaded processor when executing one single task 100% of the time; IPC_2 quantifies the average performance when executing two tasks during all the simulation, and so on.

Since the objective is to obtain a single performance value (from now on IPC_H) as a function of $sMTD$, individual IPC_i values must be combined by assigning weighting factors to each one. These factors, called hereafter ω_i , are likewise a function of $sMTD$, and determine how strongly each IPC_i must contribute to the computation of IPC_H for a given task parallelism. IPC_H can be defined as a weighted harmonic mean, given by the following equation:

$$IPC_H(sMTD) = \frac{1}{\frac{\omega_1(sMTD)}{IPC_1} + \frac{\omega_2(sMTD)}{IPC_2} + \dots + \frac{\omega_n(sMTD)}{IPC_n}}$$

This equation will be used in Section 4 to depict the performance achieved by a given design under different conditions of task parallelism on the system.

The weighting functions ω_i must fulfill the following conditions to be consistent with the $sMTD$ definition:

- i) IPC_1 must be weighted to 1 for $sMTD = 0$. The reason is that an $sMTD = 0$ represents a system running 1 task all the time, and thus, the performance of this system is specified just by IPC_1 . Mathematically, $\omega_1(0) = 1$ and $\omega_i(0) = 0$ for $i \neq 1$.
- ii) Symmetrically, IPC_n must be weighted to 1 for $sMTD = 1$. Mathematically, $\omega_n(1) = 1$ and $\omega_i(1) = 0$ for $i \neq n$.
- iii) Finally, if the system is steady, at most two IPC values can be combined with a weight other than 0, and they must be consecutive (e.g. IPC_2 and IPC_3). Mathematically, $\#\omega_i, \omega_j | \omega_i \neq 0 \wedge \omega_j \neq 0 \wedge i - j > 1$.

For a generic (non-steady) n -threaded processor with $n > 2$, it is possible to find different weighting functions that comply with conditions *i* and *ii*. Nevertheless,

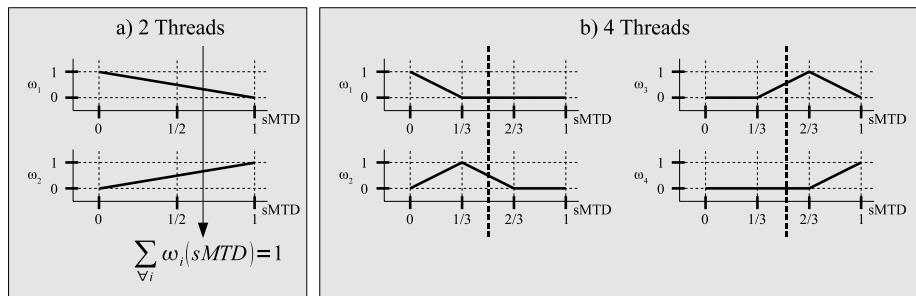


Fig. 3. Weighting functions $\omega_i(sMTD)$ in 2- and 4-threaded systems

Table 1. Baseline processor parameters

Parameter	Configuration
Machine width	8 hardware threads, 8-way fetch/issue/commit.
Storage resources	128-entry shared IQ, 64-entry shared LQ, ROBs with 32 entries per thread, per-thread 196-entry register file.
Functional units (Count/Delay)	Int.Add. (8/2), Int.Mult. (2/3), Int.Div. (2/20), Fp.Add. (2/4), Fp.Mult. (2/8), Fp.Div. (2/40).
L1 Caches (data & inst)	32KB, 2-way, 64-byte line, private per thread, 2 cycles.
L2 Cache (unified)	1MB, 8-way, 64-byte line, shared, 10 cycles.
Branch predictor	McFarling with 4K-entry gShare and 4K-entry Bimodal, 1024-entry 2-way BTB.
TLBs	16K, 4-way, shared.
Main memory	200 cycles

these functions become univocal when the additional restriction is imposed that the system load be steady (condition *iii*). Figure 3 shows a graphical representation of the weighting functions for 2- and 4-threaded steady machines. The fulfillment of condition *iii* can be observed, for example, in Figure 3b where the vertical dashed line cuts all ω_i at $sMTD = 0.5$. At this position, only ω_2 and ω_3 take a value other than 0.

Notice that, for a specific $sMTD$, the sum of the weights is always 1, which makes it unnecessary to normalize them when calculating IPC_H . The weighting functions for an 8-threaded machine are not shown, since they can be easily deduced from the conditions and examples above.

4 Experimental Results

The parameters of the modeled machine are listed in Table 1. This machine is able to fetch multiple instructions from different threads at the same cycle, by using the ICOUNT [3] fetch policy. At the dispatch stage, instructions are picked up from the fetch queue, and registers are renamed using per thread private register files. The misprediction recovery mechanism is triggered at the writeback stage, whenever a mispredicted branch is resolved.

Results shown in this section correspond to experiments with 32 ROB entries per hardware thread. Additional experiments have also been conducted with other ROB sizes, and only slight differences are observed. The simulation environment is Multi2Sim [10], a model of multicore-multithreaded processors, sharing strategies of processor resources, instruction fetch policies and thread priority assignment, among others. Benchmarks from the SPEC2000 suite have been used for the experiments. For each run, 10^8 instructions are executed from each thread, after warming up the system with another 10^8 instructions.

4.1 One-Queue ROBs

As explained in Section 2.2, one-queue ROBs have some disadvantages, such as inter-thread blocking at commit and holes at recovery. Figure 4 quantifies the

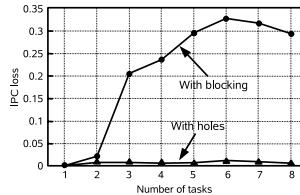


Fig. 4. Impact on IPC of inter-thread blocking and holes at recovery

negative impact on performance of these phenomena when varying the number of threads that share the same ROB.

To evaluate in an isolate manner the performance damage incurred by each phenomenon, a baseline performance value (IPC_{ideal}) has been obtained from a set of simulations of an ideal one-queue ROB. The inter-thread blocking effect is removed from this model by improving the commit logic so that it can select the oldest instruction of a thread regardless of its location. Additionally, the ROB is instantaneously collapsed when a slot is squashed, preventing holes at recovery. Eight different IPC_{ideal} values are obtained for simulations running from one up to eight tasks on the 8-threaded baseline machine.

Next, eight different performance values ($IPC_{blocking}$) are obtained, following the same procedure, and using a model of a semi-ideal one-queue ROB that does not hide the effect of inter-thread blocking. The curve shown in Figure 4 labeled *with blocking* represents the performance loss that the inter-thread blocking effect causes on the one-queue ROB machine, by means of the following equation:

$$IPC_{loss} = 1 - \frac{IPC_{blocking}}{IPC_{ideal}}$$

Likewise, the curve labeled *with holes* plots the IPC loss of the semi-ideal one-queue ROB machine that only suffers from the negative impact on performance that *holes at recovery* incur. This curve is obtained by extracting the IPC_{holes} performance value of an additional set of simulations, and using the corresponding analogous IPC_{loss} equation.

Results show that *inter-thread blocking* originates a performance loss of almost 21% when three tasks are active in the system, but this penalty is reduced to less than 3% for two tasks sharing one ROB. On the other hand, the curve corresponding to the *holes at recovery* effect shows only a slight performance degradation for any number of tasks running on the system, mainly due to small misprediction rates and fast holes draining.

4.2 ROB Sharing Strategies

Figure 5 shows the performance achieved by private, one-queue and paired ROBs on steady multithreaded processors, ranging the *sMTD* from 0 to 1, and exploring 2-, 4-, and 8-threaded systems. This range represents a wide set of commercial products that implement different number of hardware threads [11][12]. The

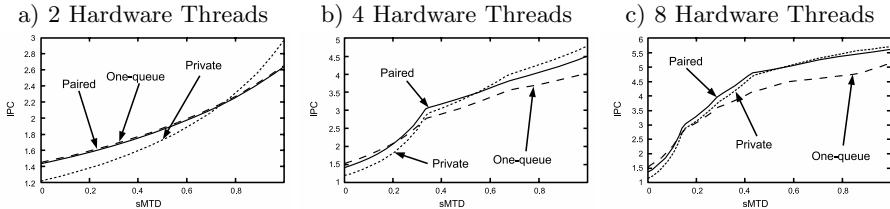


Fig. 5. ROB sharing strategies for different $sMTD$ values

performance values shown here are computed by means of the $IPC_H(sMTD)$ function, by following the steps presented in Section 3.1.

In 2-threaded processors (Figure 5a), whose load is always steady by definition, a paired ROB groups both threads in a single ROB, so it is equivalent to the one-queue ROB, as observed in the overlapped curves. When a 2-threaded system is tested using a single task ($sMTD = 0$), performance increases by more than 20% using the one-queue or paired ROB compared with the private approach. On the opposite extreme, an $sMTD = 1$ makes private ROBs outperform the other sharing strategies by 12%.

We can also appreciate that the curves corresponding to all three sharing strategies cut each other at an $sMTD$ of approximately 0.8, meaning that a larger $sMTD$ is needed to make private ROBs stand out. In other words, private ROBs are only preferable in terms of performance when two tasks are running in the system for more than 80% of the time. This occurs frequently in systems aimed at high performance computing, but it is not as common in personal computers, where multithreading is being also implemented.

In a steady 4-threaded processor (Figure 5b), the following observations can be made. In the $sMTD$ range $[0...0.35]$, the one-queue ROB performs better than private ROBs, but this behavior is inverted for the rest of $sMTD$ values. The advantages of paired ROBs can be clearly noticed in this figure. First, we can see that paired ROBs constitute the best approach for $sMTD = [0.2...0.55]$. Moreover, it performs 16% and 13% better for extreme $sMTDs$ (i.e., 0 and 1, respectively) compared to the worst approach in each case (i.e., private and one-queue, respectively). The performance loss compared to the best approach in each case is only 7% and 5%, respectively. These observations show paired ROBs as a trade-off solution to improve single-thread performance in lightly loaded multi-threaded environments, while exploiting thread level parallelism when multiple tasks run on the system.

The steady 8-threaded processor (Figure 5c) shows similar results as the 4-threaded processor, with an additional advantage for paired ROBs. Namely, the range of $sMTD$ values in which paired ROBs are the most appropriate solution grows to $[0.15...0.58]$. Again, paired ROBs show up as a good trade-off ROB sharing approach, which scales with the number of hardware threads in the system.

5 Related Work

Two previous works [7][13] compare the performance of an aggressive shared ROB, i.e., not affected by *inter-thread blocking* and *holes at recovery*, versus private ROBs. Both papers conclude that the small performance benefits obtained by these aggressive shared ROBs do not justify their higher implementation complexity.

Other related works that study the impact of distributing the resources of the SMT processor (including the ROB) assume a fixed sharing strategy. Sharkey et al. [4] propose a dynamic private ROB sharing strategy that avoids thread starvation by not allocating ROB entries of memory bounded threads. El-Moursy et al. [14] use a private ROB configuration to explore the optimal resource partitioning of a Clustered Multi-Threaded (CMT) processor. In [5] and [6] an aggressive shared ROB is distributed among the threads. In [5], Cazorla et al. dynamically classify each thread by using a set of counters to properly assign processor resources. In [6], Choi and Yeung propose a learning-based resource partitioning strategy which relies on performance metrics that can be obtained offline or during execution.

Finally, some research efforts [15][16] are devoted to dynamically assign threads to the back-ends of a given clustered multithreaded architecture. In [15], each backend can only support one thread at a given execution time, although thread switching is allowed by sharing the ROB temporarily. In [16], the back-ends are heterogeneous and multithreaded. However, each thread has its own private ROB.

6 Conclusions

In this paper, we have proposed paired ROBs as a new sharing strategy of the ROB in SMT processors. By considering the number of available software tasks allocated to hardware threads, we have developed an evaluation methodology to compare paired ROBs with the fully private and fully shared approaches. As a trade-off solution, paired ROBs minimize the effects known as inter-thread blocking, holes at recovery and starvation, while conferring flexibility to allocate ROB entries.

Experimental results show that paired ROBs provide higher performance than private ROBs for *sMTD* values lower than 0.5 in 4- and 8-threaded processors. For 2-threaded machines, paired ROBs outperform private ROBs for an *sMTD* up to 0.8. In all cases, the implementation of paired ROBs has similar complexity than private ROBs. This fact makes paired ROBs a cost-effective ROB sharing scheme.

Acknowledgements

This work is supported by Spanish CICYT under Grant TIN2006-15516-C04-01, by CONSOLIDER-INGENIO 2010 under Grant CSD2006-00046, and by Universidad Politécnica de Valencia under Grant PAID-06-07-20080029.

References

1. Tullsen, D., Eggers, S., Levy, H.: Simultaneous Multithreading: Maximizing On-Chip Parallelism. In: Proc. of the 22nd Annual International Symposium on Computer Architecture (1995)
2. El-Moursy, A., Albonesi, D.H.: Front-End Policies for Improved Issue Efficiency in SMT Processors. In: Proc. of the 9th International Conference on High Performance Computer Architecture (February 2003)
3. Tullsen, D.M., Eggers, S.J., Emer, J.S., Levy, H.M., Lo, J.L., Stamm, R.L.: Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In: Proc. of the 23rd Annual International Symposium on Computer Architecture (May 1996)
4. Sharkey, J., Balkan, D., Ponomarev, D.: Adaptive Reorder Buffers for SMT Processors. In: Proc. of the 15 International Conference on Parallel Architectures and Compilation Techniques, pp. 244–253 (2006)
5. Cazorla, F.J., Ramírez, A., Valero, M., Fernández, E.: Dynamically Controlled Resource Allocation in SMT Processors. In: Proc. of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (December 2004)
6. Choi, S., Yeung, D.: Learning-Based SMT Processor Resource Distribution via Hill-Climbing. In: Proc. of the 33rd Annual International Symposium on Computer Architecture (June 2006)
7. Raasch, S.E., Reinhardt, S.K.: The Impact of Resource Partitioning on SMT Processors. In: Proc. of the 12th International Conference on Parallel Architectures and Compilation Techniques (October 2003)
8. Sharkey, J., Ponomarev, D.V.: Efficient Instruction Schedulers for SMT Processors. In: Proc. of the 12th International Symposium on High-Performance Computer Architecture (February 2006)
9. Yeager, K.C.: The MIPS R10000 Superscalar Microprocessor. IEEE Micro. 16(2), 28–41 (1996)
10. Ubal, R., Sahuquillo, J., Petit, S., López, P.: Multi2Sim: A Simulation Framework to Evaluate Multicore-Multithreaded Processors. In: Proc. of the 19th International Symposium on Computer Architecture and High Performance Computing (October 2007), <http://www.multi2sim.org>
11. Intel Pentium Processor Extreme Edition (4 threads), <http://www.intel.com>
12. SPARC Enterprise T5240 (8 threads/core), <http://www.fujitsu.com/sparcenterprise>
13. Liu, C., Gaudiot, J.L.: Static Partitioning vs Dynamic Sharing of Resources in Simultaneous Multithreading Microarchitectures. In: Cao, J., Nejdl, W., Xu, M. (eds.) APPT 2005. LNCS, vol. 3756, pp. 81–90. Springer, Heidelberg (2005)
14. El-Moursy, A., Garg, R., Albonesi, D.H., Dwarkadas, S.: Partitioning Multi-Threaded Processors with a Large Number of Threads. In: Proc. of the IEEE International Symposium on Performance Analysis of Systems and Software (March 2005)
15. Latorre, F., González, J., González, A.: Back-end Assignment Schemes for Clustered Multithreaded Processors. In: Proc. of the 18th Annual international Conference on Supercomputing (June 2004)
16. Acosta, C., Falcon, A., Ramírez, A.: A Complexity-Effective Simultaneous Multithreading Architecture. In: Proc. of the 2005 International Conference on Parallel Processing (June 2005)

REPAS: Reliable Execution for Parallel ApplicationS in Tiled-CMPs[∗]

Daniel Sánchez, Juan L. Aragón, and José M. García

Departamento de Ingeniería y Tecnología de Computadores

Universidad de Murcia, 30071 Murcia (Spain)

{dsanchez, jlaragon, jmgarcia}@ditec.um.es

Abstract. Reliability has become a first-class consideration issue for architects along with performance and energy-efficiency. The increasing scaling technology and subsequent supply voltage reductions, together with temperature fluctuations, augment the susceptibility of architectures to errors. Previous approaches have tried to provide fault tolerance. However, they usually present critical drawbacks concerning either hardware duplication or performance degradation, which for the majority of common users results unacceptable.

RMT (Redundant Multi-Threading) is a family of techniques based on SMT processors in which two independent threads (master and slave), fed with the same inputs, redundantly execute the same instructions, in order to detect faults by checking their outputs. In this paper, we study the under-explored architectural support of RMT techniques to reliably execute shared-memory applications. We show how atomic operations induce to serialization points between master and slave threads. This bottleneck has an impact of 34% in execution time for several parallel scientific benchmarks. To address this issue, we present REPAS (Reliable execution of Parallel ApplicationS in tiled-CMPs), a novel RMT mechanism to provide reliable execution in shared-memory applications.

While previous proposals achieve the same goal by using a big amount of hardware - usually, twice the number of cores in the system - REPAS architecture only needs a few extra hardware, since the redundant execution is made within 2-way SMT cores in which the majority of hardware is shared. Our evaluation shows that REPAS is able to provide full coverage against soft-errors with a lower performance slowdown in comparison to a non-redundant system than previous proposals at the same time it uses less hardware resources.

1 Introduction

The increase in the number of available transistors in a chip has made it possible to build powerful processors. In this way, CMPs have become a good approach

[∗] This work has been jointly supported by the Fundación Séneca (Agencia Regional de Ciencia y Tecnología, Región de Murcia) under grant 05831/PI/07, also by the Spanish MEC and European Commission FEDER funds under grants “Consolider Ingenio-2010 CSD2006-00046” and “TIN2006-15516-C04-03”.

to improve performance in an energy-efficient way, while keeping a manageable complexity to exploit the thread-level parallelism. Furthermore, for current and future CMP architectures, more efficient designs are tiled-CMPs [18]. These architectures are organized around a direct network, since area, scalability and power constraints make impractical the use of a bus as the interconnection network.

However, this trend in the increasing number of transistors per chip has a major drawback due to the growth of the failure ratio in every new scale generation [14]. Firstly, the fact of having a higher number of transistors per chip increases the probability of a fault. Secondly, the growth of temperature, the decrease of the supply voltage and the subthreshold voltage in the chip, in addition to other non-desirable effects such as higher power supply noise, signal cross-talking, process variation or in-progress wear-out, compromise the system's reliability.

Hardware errors can be divided into three main categories: transient faults, intermittent faults and permanent faults [8, 3]. Both transient and intermittent faults appear and disappear by themselves. The difference is that, while transient faults are originated by external agents like particle strikes over the chip, intermittent faults are caused by intra-chip factors, such as process variation combined with voltage and temperature fluctuations [20]. In addition, transient faults disappear faster than intermittent faults, on average. Finally, permanent faults remain in the hardware until the damaged part is replaced. Therefore, this paper aims at the study of fault tolerant techniques to detect and recover from transient and intermittent faults, also known as soft errors.

Although the fault ratio does not represent a handicap for the majority of users, several studies show how soft errors can heavily damage industry [8]. For instance, in 1984 Intel had certain problems delivering chips to AT&T as a result of alpha particle contamination in the manufacturing process. In 2000, a reliability problem was reported by Sun Microsystems in its UltraSparc-II servers deriving from insufficient protection in the SRAM. A report from Cypress Semiconductor shows how a car factory is halted once a month because of soft errors [21]. Another fact to take into account is that the fault ratio increases due to altitude. Therefore, reliability has become a major design problem in the aerospace industry.

Fault detection has usually been achieved through the redundant execution of the program instructions, while recovery methods are commonly based on checkpointing. A checkpoint reflects a safe state of the architecture in a temporal point. When a fault is detected, the architecture is rolled-back to the previous checkpoint and the execution is restarted.

There are several proposals to achieve fault tolerance in microarchitectures. RMT (Redundant Multi-Threading) is a family of techniques based on redundant execution in which two threads execute the same instructions. Simultaneous and Redundantly Threaded processors (SRT) [12] and SRT with Recovery (SRTR) [19] are two of them, implemented on SMT processors in which two independent and redundant threads are executed with a delay respect to the other which

speeds up their execution. These early approaches are attractive since they do not require many design changes in a traditional SMT processor. In addition, they only add some extra hardware for communication purposes between the threads. However, the major drawback of SRT(R) is the inherent non-scalability of SMT processors as the number of threads increases. The arrival of CMP architectures contributed to solve this problem, with proposals such as Chip-level Redundant Threaded processors (CRTR) [2], Reunion [16] and Dynamic Core Coupling (DCC) [5]. These techniques, in contrast, suffer from high inter-core communication through either dedicated buses or the underlying interconnection network.

Although there are different approaches using SRTR and CRTR with sequential or independent multithreaded applications [19][4], the architectural support for redundant execution with shared-memory workloads is not well suited, since atomic operations induce a serialization point between master and slave threads, as we will show in Section 3.1. On the other hand, more recent proposals such as Reunion or DCC, aimed at parallel workloads, use a shared-bus as interconnection network. Therefore, when they are moved to a more scalable environment such as a direct-network, they suffer from significant performance degradation as the number of cores increases (as studied in [17]), in addition to the extra number of cores that they require.

To address all these issues, in this paper we propose *Reliable Execution of Parallel ApplicationS* (REPAS) in tiled-CMPs. The main features that characterize REPAS are:

- A scalable solution built on adding dual SMT cores to form a tiled-CMP.
- Ability to detect and recover from transient and intermittent faults.
- Reduced performance overhead as compared to previous proposals (less than 13% slowdown than CRTR and 7% less than DCC for 16-thread workloads. Recall that DCC uses twice the number of cores (i.e., 32 *vs.* 16 cores) for those reported slowdowns.
- Low decoupling between master and slave threads which allows both quicker fault detection and faster recovery times (reduced to just tens of cycles). The former represents an interesting feature for critical applications in which fault detection cannot be delayed long; being the latter an important feature in case of a burst of faults, as it usually occurs for intermittent faults [20].

The rest of the paper is organized as follows. Section 2 provides some background and reviews related work. Section 3 details the previous CRTR architecture and shows its performance issues in shared-memory environments. In Section 4 we describe REPAS and Section 5 shows the experimental results. Finally, Section 6 summarizes the main conclusions of our work.

2 Related Work

When comparing different fault-tolerant mechanisms, we can point out four main characteristics. Firstly, the *sphere of replication or SoR* [12], which determines the components in the microarchitecture that are replicated. Secondly,

Table 1. Main characteristics of several fault-tolerant architectures

	SoR	Synchronization	Input Replication	Output Comparison
SRT(R) CRT(R)	Pipeline, Registers	Staggered execution	Strict (Queue-based)	Instruction by instruction
Reunion	Pipeline, Registers, L1Cache	Loose coupling	Relaxed input replication	Fingerprints
DCC	Pipeline, Registers, L1Cache	Thousands of instructions	Consistency window	Fingerprints, Checkpoints
HDTLR	Pipeline, Registers, L1Cache	Thousands of instructions	Sub-epochs	Fingerprints, Checkpoints

the *synchronization*, which indicates how often redundant copies compare their computation results. Thirdly, the *input replication* method, which defines how redundant copies observe the same data. Finally, the *output comparison* method, which defines how the correctness of the computation is assured. Table 1 summarizes the main characteristics of the proposals covered in this section.

One of the first approaches to full redundant execution is Lockstepping [1], a proposal in which two statically bound execution cores receive the same inputs and execute the same instructions step by step. Later, the family of techniques Simultaneous and Redundantly Threaded processors (SRT) [12], SRT with Recovery (SRTR) [19], Chip-level Redundantly Threaded processors (CRT) [9] and CRT with Recovery (CRTR) [2] was proposed, based on a previous approach called AR-SMT [13]. In SRT(R) redundant threads are executed within the same core. The SoR includes the entire SMT pipeline but the first level of cache. The threads execute in a *staggered execution* mode, using strict input replication and output comparison on every instruction. Other studies have chosen to allocate redundant threads in separate cores. This way, if a permanent fault damages an entire core, a single thread can still be executed. Among these studies it is worth mentioning CRT(R) [9, 2], Reunion [16], DCC [5] and HDTLR [11]. In all these proposals, a fundamental point is how redundant pairs communicate with each other, as we will summarize later.

In Reunion, the *vocal core* is responsible for accessing and modifying shared-memory coherently. However, the *mute core* only accesses memory by means of non-coherent requests called *phantom requests*, providing redundant access to the memory system. This approach is called *relaxed input replication*. In order to detect faults, the current architectural state is interchanged among redundant cores by using a compression method called *fingerprinting* [15] through a dedicated point-to-point fast bus. Relaxed input replication leads to input incoherence which are detected as faults. As a result, checking intervals must be short (hundred of instructions) to avoid excessive penalties. Violations in relaxed input replication induce to a serialized execution (very similar to lock-stepped execution) between redundant cores, affecting performance with a degradation of 22% over a base system when no faults are injected.

Dynamic Core Coupling (DCC) [5] does not use any special communication channel and reduces the overhead of Reunion by providing a decoupled execution of instructions, making larger comparison intervals (thousand of instructions) and reducing the network traffic. At the end of each interval, the state of

redundant pairs is interchanged and, if no error is detected, a new checkpoint is taken. As shown in [5], the optimal checkpoint interval for DCC is 10,000 cycles, meaning that the time between a fault happens and its detection is usually very high. Input incoherences are avoided by a consistency window which forbids data updates, while the members of a pair have not observed the same value. However, DCC uses a shared bus as interconnection network, which simplifies the consistency window mechanism. Nevertheless, this kind of buses are not scalable due to area and power constraints. In [17], DCC is studied in a direct-network environment, and it is shown that the performance degradation rises to 19%, 39% and 42% for 8, 16, and 32 core pairs.

Recently, Rashid et al. proposed Highly-Decoupled Thread-Level Redundancy (HDTLR) [11]. HDTLR architecture is similar to DCC, in which the recovery mechanism is based on checkpoints which reflect the architecture changes between *epochs*, and modifications are not made visible to L2 until verification. However, in HDTLR each redundant thread is executed in different hardware contexts (*computing waveform* and *verification waveform*), maintaining coherency independently. This way, the consistency window is avoided. However, the asynchronous progress of the two hardware contexts could lead to memory races, which result in different execution outcomes, masking this issue as a transient fault. In a worst-case scenario, not even a rollback would guarantee forward progress. Thus, an order tracking mechanism, which enforces the same access pattern in redundant threads, is proposed. This mechanism implies the recurrent creation of sub-epochs by expensive global synchronizations. Again, in this study the interconnection network is a non-scalable shared-bus.

3 CTRR as a Building Block for Reliability

CTR is a fault tolerance architecture proposed by Gomaa et al. [2], an extension to SRTR [19], for CMP environments. In CTR, two redundant threads are executed on separate SMT processor cores, providing transient fault detection. Furthermore, since redundant threads are allocated in distant hardware, the architecture is, as well, potentially able to tolerate permanent faults. These threads are called *master* (or *leading*) and *slave* (or *trailing*) threads, since one of them runs ahead the other by a number of instructions called *slack*. As in a traditional SMT processor, each thread owns a PC register, a renaming map table and a register file, while all the other resources are shared.

The master thread is responsible for accessing memory to load data which bypasses to the slave thread, along with the accessed address. Both data and addresses are kept in a FIFO structure called Load Value Queue (LVQ) [12]. This structure prevents the slave thread from observing different values from those the master did, a phenomenon called *input incoherence*. The aim in CTR is to avoid cache updates until the corresponding value has been verified. In order to do that, when a store instruction commits in the master, the value and accessed address are bypassed to the slave which keeps them in a structure called

Store Value Queue (SVQ) [12]. When a store commits in the slave, it accesses the SVQ and if the check successes, the L1 cache is updated. Other structures used in CRTR are the Branch Outcome Queue (BOQ) [12] and the Register Value Queue (RVQ) [19]. The BOQ is used by the master to bypass the destination of a branch to the slave which uses this information as branch predictions. Availability for these hints is assured thanks to the slack, since by the time the slave needs to predict a branch, the master knows the correct destination, which bypasses to the slave. This way, the execution speed of the latter is increased because branch mispredictions are avoided. Finally, the RVQ is used to bypass register values of every committed instruction by the master, which are needed for checking. Whenever a fault is detected, the recovery mechanism commences. The slave register file is a safe point, since no updates are performed on it until a successful verification. Therefore, the slave bypasses the content of its register file to the master, pipelines of both threads are flushed and execution is restarted from the detected faulty instruction.

As it was said before, separating the execution of a master thread and its corresponding slave in different cores adds the ability to tolerate permanent faults. However, it requires a wide datapath between cores in order to bypass all the information required for checking. Furthermore, although wire delays may be hidden by the slack, the cores bypassing data must be close to each other to avoid stalling.

3.1 CRTR in Shared-Memory Environments

Although CRTR was originally proposed and evaluated for sequential applications [9, 2], the authors argue that it could be used for multithreaded applications, too. However, we have found that, with no additional restrictions, CRTR can lead to wrong program execution for shared-memory workloads in a CMP scenario, even in the absence of transient faults.

In shared-memory applications, such as those that can be found in SPLASH-2, the access to critical sections is granted by primitives which depend on atomic instructions. In CRTR, the master thread never updates memory. Therefore, when a master executes the code to access a critical section, the value of the variable “lock” will not be visible until the slave executes and verifies the same instructions. This behaviour enables two (or more) master threads to access a critical section at the same time, which potentially leads to a wrong execution.

In order to preserve sequential consistency and, therefore, the correct program execution, the most straightforward solution is to synchronize master and slave threads whenever an atomic instruction is executed. This conservative approach introduces a noticeable performance degradation (an average 34% slowdown as evaluated in Section 5.2). The duration of the stall of the master thread depends on two factors: (1) the size of the slack, which determines how far the slave thread is, and (2) the number of write operations in the SVQ, which must be written in L1 prior to the atomic operation to preserve consistency.

4 REPAS Architecture

At this point, we propose *Reliable Execution for Parallel ApplicationS* (REPAS) in tiled-CMPs. We create the reliable architecture of REPAS by adding CRTR cores to form a tiled-CMP. However, we avoid the idea of separating master and slave threads in different cores. We justify this decision for two main reasons: (1) as a first approach, our architecture will not tolerate permanent faults whose appearance ratio is still much lower than the ratio of transient faults [10], and (2) we avoid the use of the expensive inter-core datapaths. An overview of the core architecture can be seen in Figure 1. As in a traditional SMT processor, issue queues, register file, functional units and L1-cache are shared among the threads. The shaded boxes in Figure 1 represent the extra hardware introduced by CRTR and REPAS as explained in Section 3.

In benchmarks with high contention resulting from synchronization, the constraint described in Section 3.1 for CRTR, may increase the performance degradation of the architecture dramatically. To avoid frequent master stalls derived from consistency, we propose an alternative management of stores. Instead of updating memory just after verification, a more suitable approach is allowing updates in L1 cache without checking. By doing so, the master thread will not be stalled as a result of synchronizations.

Collaterally, we clearly reduce the pressure on the SVQ. In the original CRTR implementation, a master's load must look into the SVQ to obtain the value produced by an earlier store. This implies an associative search along the structure for every load instruction. In REPAS, we eliminate these searches since the up-to-date values for every block are stored in L1 cache where they can be accessed as usual.

Unfortunately, this measure complicates the recovery mechanism. When a fault is detected, the L1 cache may have unverified blocks. Upon detection, all unverified blocks must be invalidated. Furthermore, when a store is correctly checked by the slave, the word in L1 must be written-back into L2. This way, the L2 cache remains consistent, even if the block in L1 is invalidated as a

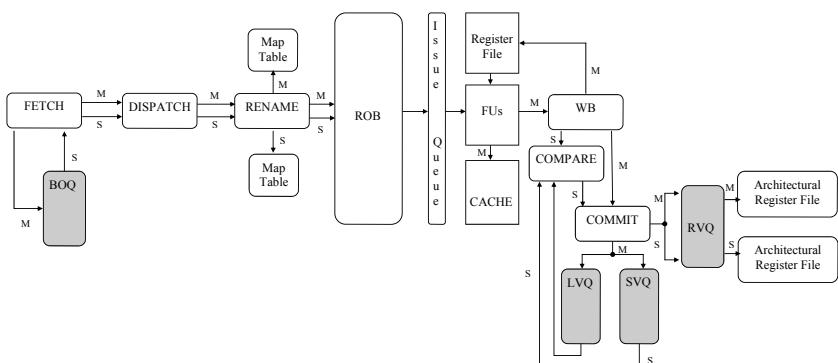


Fig. 1. REPAS core architecture overview

result of a fault. This mechanism is carried out in background, and despite the increased L1-to-L2 traffic, it does not have a noticeable impact on performance.

4.1 Implementation Details of REPAS

To avoid error propagation deriving from a wrong result stored in L1 cache by the master, unverified blocks in cache must be identified. In order to do this, we introduce an additional bit per L1 cache block called *Unverified bit* which is activated on any master write. When the Unverified bit is set on a cache block, it cannot be displaced or shared with other nodes, effectively avoiding the propagation of a faulty block. Eventually, the Unverified bit will be cleared when the corresponding slave thread verifies the correct execution of the memory update.

Clearing the Unverified bit is not a trivial task. A master thread can update a cache block several times before a verification takes place. If the first check performed by the slave is successful, it means that the first memory update was valid. However, this does not imply that the whole block is completely verified, since the rest of the updates has not been checked yet. One simple way of knowing if a block needs more checks before clearing the unverified bit is by looking if the block appears more than once in the SVQ. If it does, more verifications need to be performed. Yet, this measure implies an associative search in the SVQ. Nonetheless, as we said before, we eliminate much of the pressure produced by master's loads. In quantitative terms, in the original CRTR proposal there was an associative search every master's load, and now we have an associative search every slave's store. This results in a significant reduction of associative searches within the SVQ, given the fact that the load/store ratio for the studied benchmarks is almost 3 to 1. Furthermore, as this operation is performed in parallel to the access to L1 cache, we do not expect an increase in the cycle time to access L1-cache.

5 Evaluation

5.1 Simulation Environment

We have implemented REPAS as well as the proposed extensions to CRTR to support the execution of parallel applications, evaluating the performance results by using the functional simulator Virtutech Simics [6], extended with the execution-driven multiprocessor simulator GEMS [7].

Our study has been focused on a 16-way tiled-CMP in which each core is a dual-threaded SMT, which has its own private L1 cache, a portion of the shared L2 cache and a connection to the on-chip network. The architecture follows the sequential consistency model with the write-read reordering optimization. The main parameters of the architecture are shown in Table 2(a). For the evaluation, we have used a selection of scientific applications: Barnes, FFT, Ocean, Radix, Raytrace, Water-NSQ and Water-SP are from the SPLASH-2 benchmark suite.

Table 2. Characteristics of the evaluated architecture and used benchmarks

(a) System characteristics			
16-Way Tiled CMP System		Cache Parameters	
Processor Speed	2 GHz	Cache line size	64 bytes
Max. Fetch / retire rate	4 instructions / cycle	L1 cache	
Consistency model	Sequential Consistency	Size, associativity	64KB, 4 ways
Memory parameters		Hit time	1 cycle
Coherence protocol	MOESI	Shared L2 cache	
Write Buffer	64 entries	Size, associativity	512KB/tile, 4 ways
Memory access time	300 cycles	Hit time	15 cycles
Network parameters		Fault tolerance parameters	
Topology	2D mesh	LVQ/SVQ	64 entries each
Link latency (one hop)	4 cycles	RVQ	80 entries
Flit size	4 bytes	BOQ	64 entries
Link bandwidth	1 flit/cycle	Slack Fetch	256 instructions

(b) Input sizes			
Benchmark	Size	Benchmark	Size
Barnes	8192 bodies, 4 time steps	Tomcatv	256 points, 5 iterations
FFT	256K complex doubles	Unstructured	Mesh.2K, 5 time steps
Ocean	258 x 258 ocean	Water-NSQ	512 molecules, 4 time steps
Radix	1M keys, 1024 radix	Water-SP	512 molecules, 4 time steps
Raytrace	10Mb, teapot.env scene		

Tomcatv is a parallel version of a SPEC benchmark and Unstructured is a computational fluid dynamics application. The experimental results reported in this work correspond to the parallel phase of each program only. Sizes problems are shown in Table 2(b)

5.2 Performance Analysis

We have simulated the benchmarks listed in Table 2(b) in a tiled-CMP with 16 cores. Figure 2 compares CRTR with REPAS. The results are normalized to a system in which no redundancy is introduced. Overall, these results clearly show that REPAS performs better than CRTR. This tendency is more noticeable in benchmarks such as Unstructured or Raytrace, which present many more atomic synchronizations than the rest of the studied benchmarks, as it can be observed in Table 3.

CRTR obtains an average performance degradation of 34% in comparison to a non-redundant system. On the contrary, REPAS is able to reduce this degradation down to 21%. Atomic operations damage CRTR in two ways. Firstly, they act as serialization points: the slave thread must catch up with the master. Secondly, all the stores in the SVQ must be issued to memory before the actual atomic operation, in order to preserve the sequential consistency model.

Table 3. Frequency of atomic synchronizations (per 100 cycles)

	Barnes	FFT	Ocean	Radix	Raytrace	Tomcatv	Unstructured	Water-NSQ	Water-SP
Synchronizations	0.162	0.039	0.142	0.013	0.2	0.02	3.99	0.146	0.014
Cycles per synchronization	478.7	405.1	376.7	451.2	561.9	405.6	566	563.7	408.8

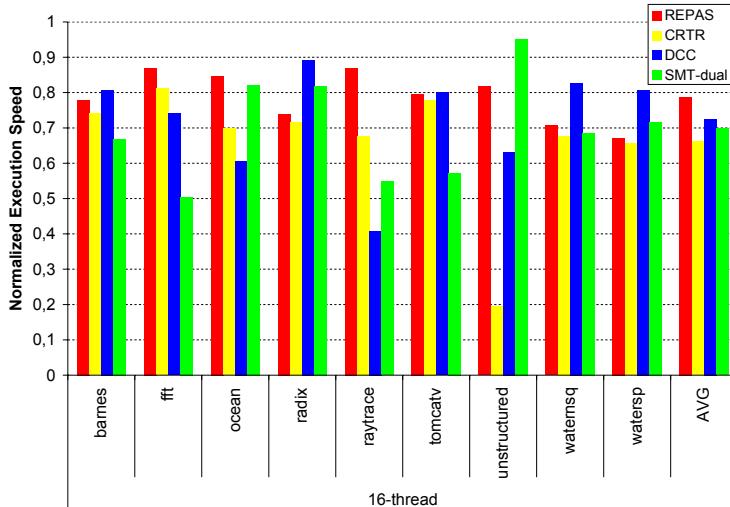


Fig. 2. Execution time overhead over a non fault-tolerant architecture

We have also evaluated DCC in a 32-core CMP with a 2D-mesh network, using the same parameters shown in Table 2(a). As studied in [17], DCC performs poorly in this environment due to the latency imposed by the *age table*, introduced to maintain the master-slave consistency. As we can see in Figure 2, REPAS is 7% faster than DCC on average. However, it is important to note that DCC uses twice as much hardware as REPAS, since the redundant threads are executed in different cores. This represents another advantage of REPAS over DCC.

Finally, we show the performance of SMT-dual, a coarse-grained redundancy approach which represents a 16-core 2-way SMT architecture executing two copies (A and A') of each studied application. Within each core, one thread of A and one thread of A' are executed. As mentioned in [12], this helps to illustrate what the performance degradation of a SMT processor is when two copies of the same thread are running within the same core. As we can see, REPAS is 9% faster than SMT-dual on average which, at the same time, is faster than CRTR in 2%.

5.3 Speculative Sharing

REPAS does not allow the sharing of unverified blocks as a conservative measure to avoid the propagation of errors among cores. On the contrary, DCC [5] is based on a speculative sharing policy. Given that blocks are only verified at checkpointing creation intervals (i.e., 10,000 cycles), avoiding speculative sharing in DCC would degrade performance in an unacceptable way.

For comparison purposes, we have studied the effect of sharing unverified blocks in REPAS. In order to avoid an unrecoverable situation, speculatively delivered data block the commit stage of the requestor. In this way, we introduce

Table 4. Number of speculative sharings and time needed to verify those blocks

	Barnes	FFT	Ocean	Radix	Raytrace	Tomcatv	Unstructured	Water-NSQ	Water-SP	Avg
# Speculative sharings	12077	50	9217	901	39155	163	224286	244	252	-
Time to verification	102	91.72	96.71	81.8	80.79	107.337	113.14	102.377	76.76	95

new data in the pipeline to operate with (similarly to conventional speculative execution to support branch prediction). Furthermore, a speculative block can be shared by two or more requestors. When the producer validates the block, it sends a signal to all the sharers confirming that the acquired block was correct and the commit stage is re-opened in their pipelines. If a core which produced speculatively shared data detects a fault, an invalidation message is sent to all the sharers in order to flush their pipelines, undoing the previous work.

We have not considered to migrate unverified data speculatively, since an expensive mechanism would be necessary to keep track of the changes in the ownership, the sharing chains as well as in the original value of the data block for recovery purposes.

Table 4 reflects that speculations are highly uncommon, and that all the time in which we could benefit from, 95 cycles on average, cannot be fully amortized because pipeline is closed at commit. This explains why speculative sharings do not obtain much benefit in REPAS. The performance evaluation shows a negligible performance improvement on average. However, for those benchmarks as Barnes, Raytrace and Unstructured which show a greater number of sharings, the performance is increased around 1% over the basic mechanism. Overall, the performance increase due to speculative sharing seems inadequate, since it is not worth the incremented complexity in the recovery mechanism of the architecture.

5.4 Transient Fault Injection

We have shown that REPAS is able to reduce the performance degradation of previous proposals in a fault-free scenario. However, faults and the recovery mechanism to solve them introduce an additional degradation which must be also studied. Figure 3 shows the execution overhead of REPAS under different fault rates. Failure rates are expressed in terms of faulty instructions per million of cycles per core. These rates are much higher than expected. However, they are being evaluated to overstress the architecture and to show the scalability of the system.

As we can see in Figure 3(a), REPAS is able to tolerate rates of 100 faulty instructions per million per core with an average performance degradation of 2% over a test with no injected faults. The average overhead grows to 10% when the fault ratio is increased to 1000 per million. The time spent on every recovery depends on the executed benchmark and it is 80 cycles on average, as we can see in Figure 3(b). This time includes the invalidation of all the unverified blocks and the rollback of the architecture up to the point where the fault was detected. Although we have not completely evaluated it yet, other proposals such as DCC spend thousands of cycles to achieve the same goal (10,000 cycles in a worst-case scenario). This clearly shows the greater scalability of REPAS in a faulty environment.

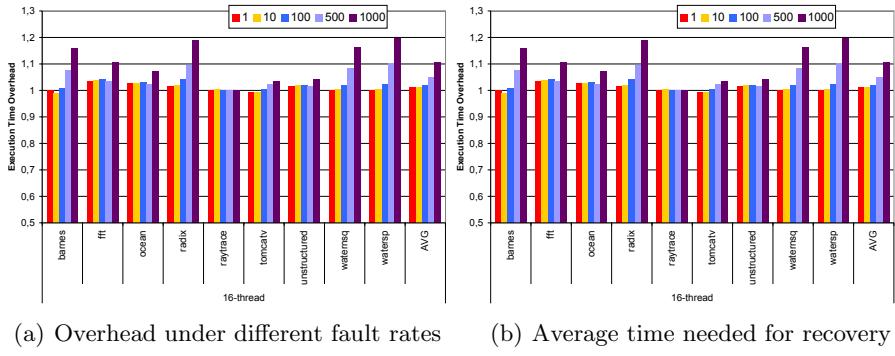


Fig. 3. Execution time overhead and rollback time under different failure rates

6 Conclusions

Processors are becoming more susceptible to transient faults due to several factors such as technology scaling, voltage reduction, temperature fluctuations, process variation or signal cross-talking. Although there are many approaches exploring reliability for single-threaded applications, shared-memory environments have not been thoroughly studied.

In this paper, we first study the under-explored architectural support for CRTR to reliably execute shared-memory applications. We show how atomic operations induce a serialization point between master and slave threads. A bottleneck which has an average impact of 34% in the execution time over several parallel scientific benchmarks.

To address this issue, we propose REPAS: Reliable Execution for Parallel ApplicationS in Tiled-CMPs, where we allow updates in L1 cache before verification. Thus, we obtain a more decoupled execution, reducing the stall time due to synchronization. To avoid fault propagation among cores, unverified data reside in L1 cache, in which sharing is not allowed as a conservative measure. With this mechanism, we can reduce the overall performance degradation to 21% with regards to a non-redundant system, advantaging other recent approaches such as Dynamic Core Coupling (DCC) which has an overall impact of 28% using twice the number of cores. Finally, we have also shown that our proposal is able to tolerate fault rates up to 100 faulty instructions per million of executed instructions per core, with an overall performance overhead of hardly 2% over a system with no injected faults.

References

- [1] Bartlett, J., Gray, J., et al.: Fault tolerance in tandem computer systems. In: The Evolution of Fault-Tolerant Systems (1987)
- [2] Gomaa, M., Scarbrough, C., et al.: Transient-fault recovery for chip multiprocessors. In: Proc. of the 30th annual Int' Symp. on Computer architecture (ISCA 2003), San Diego, California, USA (2003)

- [3] González, A., Mahlke, S., et al.: Reliability: Fallacy or reality? *IEEE Micro.* 27(6) (2007)
- [4] Kumar, S., Aggarwal, A.: Speculative instruction validation for performance-reliability trade-off. In: Proc. of the 2008 IEEE 14th Int' Symp. on High Performance Computer Architecture (HPCA 2008), Salt Lake City, USA (2008)
- [5] LaFrieda, C., Ipek, E.: et al. Utilizing dynamically coupled cores to form a resilient chip multiprocessor. In: Proc. of the 37th Annual IEEE/IFIP Int' Conf. on Dependable Systems and Networks (DSN 2007), Edinburgh, UK (2007)
- [6] Magnusson, P., Christensson, M., et al.: Simics: A full system simulation platform. *Computer* 35(2) (2002)
- [7] Martin, M.K., Sorin, D.J., et al.: Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News* 33(4) (2005)
- [8] Mukherjee, S.: Architecture design for soft errors. Morgan Kaufmann, San Francisco (2008)
- [9] Mukherjee, S., Kontz, M., et al.: Detailed design and evaluation of redundant multithreading alternatives. In: Proc. of the 29th annual Int' Symp. on Computer architecture (ISCA 2002), Anchorage, AK, USA (2002)
- [10] Pizza, M., Strigini, L., et al.: Optimal discrimination between transient and permanent faults. In: Third IEEE International High-Assurance Systems Engineering Symposium, pp. 214–223 (1998)
- [11] Rashid, M., Huang, M.: Supporting highly-decoupled thread-level redundancy for parallel programs. In: Proc. of the 14th Int' Symp. on High Performance Computer Architecture (HPCA 2008), Salt Lake City, USA (2008)
- [12] Reinhardt, S.K., Mukherjee, S.: Transient fault detection via simultaneous multithreading. In: Proc. of the 27th annual Int' Symp. on Computer architecture (ISCA 2000), Vancouver, BC, Canada (2000)
- [13] Rotenberg, E.: Ar-smt: A microarchitectural approach to fault tolerance in microprocessors. In: Proc. of the 29th Annual Int' Symp. on Fault-Tolerant Computing (FTCS 1999), Madison, WI, USA (1999)
- [14] Shivakumar, P., Kistler, M., et al.: Modeling the effect of technology trends on soft error rate of combinational logic. In: Proc. of the Int' Conf. on Dependable Systems and Networks (DSN 2002), Bethesda, MD, USA (2002)
- [15] Smolens, J.C., Gold, B.T., et al.: Fingerprinting: Bounding soft-error-detection latency and bandwidth. *IEEE Micro.* 24(6) (2004)
- [16] Smolens, J.C., Gold, B.T., et al.: Reunion: Complexity-effective multicore redundancy. In: Proc. of the 39th Annual IEEE/ACM Int' Symp. on Microarchitecture (MICRO 39), Orlando, FL, USA (2006)
- [17] Sánchez, D., Aragón, J.L., et al.: Evaluating dynamic core coupling in a scalable tiled-cmp architecture. In: Proc. of the 7th Int. Workshop on Duplicating, Deconstructing, and Debunking (WDDD 2008). In conjunction with ISCA (2008)
- [18] Taylor, M.B., Kim, J., et al.: The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro.* 22(2), 25–35 (2002)
- [19] Vijaykumar, T., Pomeranz, I., et al.: Transient fault recovery using simultaneous multithreading. In: Proc. of the 29th Annual Int' Symp. on Computer Architecture (ISCA 2002), Anchorage, AK (2002)
- [20] Wells, P.M., Chakraborty, K., et al.: Adapting to intermittent faults in multicore systems. In: Proc. of the 13th Int' Conf. on Architectural support for programming languages and operating systems (ASPLOS 2008), Seattle, WA, USA (2008)
- [21] Zielger, J.F., Puchner, H.: SER-History, Trends and Challenges. Cypress Semiconductor Corporation (2004)

Impact of Quad-Core Cray XT4 System and Software Stack on Scientific Computation

S.R. Alam, R.F. Barrett, H. Jagode, J.A. Kuehn, S.W. Poole, and R. Sankaran

Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA

{alam, rbarrett, jagode, kuehn, poole, sankaran}@ornl.gov

Abstract. An upgrade from dual-core to quad-core AMD processor on the Cray XT system at the Oak Ridge National Laboratory (ORNL) Leadership Computing Facility (LCF) has resulted in significant changes in the hardware and software stack, including a deeper memory hierarchy, SIMD instructions and a multi-core aware MPI library. In this paper, we evaluate impact of a subset of these key changes on large-scale scientific applications. We will provide insights into application tuning and optimization process and report on how different strategies yield varying rates of successes and failures across different application domains. For instance, we demonstrate that the vectorization instructions (SSE) provide a performance boost of as much as 50% on fusion and combustion applications. Moreover, we reveal how the resource contentions could limit the achievable performance and provide insights into how application could exploit Petascale XT5 system's hierarchical parallelism.

1 Introduction

Scientific productivity on the emerging Petascale systems is widely attributed to the system balance in terms of processor, memory, network capabilities and the software stack. The next generations of these Petascale systems are likely to be composed of processing elements (PE) or nodes with 8 or more cores on single or multiple sockets, deeper memory hierarchies and a complex interconnection network infrastructure. Hence, the development of scalable applications on these systems cannot be achieved by a uniformly balanced system; it requires application developers to develop a hierarchical view where memory and network performance follow a regular but non-uniform access model. Even the current generation of systems with peak performance of hundreds of Teraflops such as the Cray XT and IBM Blue Gene series systems offer 4 cores or execution units per PE, multiple levels of unified and shared caches and a regular communication topology along with support for distributed computing (message-passing MPI) and hybrid (MPI and shared-memory OpenMP or pthreads) programming models [Dagnum98, Snir98, BGL05, BGP08, XT3a-b, XT4a-b]. As a result, it has become extremely challenging to sustain let alone to improve performance efficiencies or scientific productivity on the existing systems as we demonstrate in this paper. At the same time however, these systems serve as test-beds for applications targeting Petascale generation systems that are composed of hundreds of thousands of processing cores.

We have extensive experience of benchmarking and improving performance efficiencies of scientific applications on the Cray XT series systems, beginning from the first-generation, single-core AMD based ~26 Teraflops Cray XT3 system to the latest quad-core based ~263 Teraflops Cray XT4 system. During these upgrades, a number of system software and hardware features were modified and replaced altogether such as migration from Catamount to Compute Node Linux (CNL) operating system, network capabilities and support for hybrid programming models and most importantly multi-core processing nodes [Kelly05]. A complete discussion of individual features are beyond the scope of this paper, however we do attempt to provide a comprehensive overview of the features updated in the latest quad-core upgrade and how these features impact performance of high-end applications. We provide an insight by using a combination of micro-benchmarks that highlight specific features and then provide an assessment of how these features influence overall performance of complex, production-level applications and how performance efficiencies are improved on the target platform.

In this paper, we focus on micro-architectural characteristics of the quad-core system particularly the new vectorization units and the shared level 3 cache. This study also enables us to identify features that are likely to influence scientific productivity on the Petascale Cray XT5 system. Hence, a unique contribution of this paper is that it not only evaluates performance of a range of scientific applications on one of the most powerful open-science supercomputing platform but also discusses how the performance issues are addressed during the quad-core upgrade. The Cray XT5 system shares a number of features including the processor and the network infrastructure with its predecessor, the quad-core XT4 system. However, the XT5 system has some distinct characteristics; most importantly, hierarchical parallelism within a processing node since an XT5 PE is composed of two quad-core processors thereby yielding additional resource contentions for memory, network and file I/O operations.

The outline of the paper is as follows: background and motivation for this research along with a description of target system hardware and software features is provided in section 2. In section 3, we briefly outline micro-benchmarks and high-end scientific applications that are targeted for this study. Details of experiments and results relating to each feature that we focus on in this paper are presented in section 4. Conclusions and future plans are outlined in section 5.

2 Motivation and Background

The Cray XT system located at ORNL is the second most powerful computing capability for the Department of Energy's (DOE) Office of Science, and in fact represents one of the largest open science capability platforms in the United States. Named Jaguar, it is the primary leadership computer for the DOE Innovative and Novel Computational Impact on Theory and Experiment (INCITE) program, which supports computationally intensive, large-scale research projects. The 2008 program awarded over 140 million processor hours on Jaguar to groups investigating a broad set of science questions, including global climate dynamics, fusion, fission, and combustion energy, biology, astrophysics, and materials.

In order to support this scale of computing, Jaguar has been upgraded from a 119 Teraflops capability to 262 Teraflops (TFLOPS). Several fundamental characteristics of the architecture have changed with this upgrade, which have a wide-ranging impact across different application domains. This motivates our research of identifying and quantifying the impact of these new architectural and system software stack features on leadership scale applications.

The current incarnation of Jaguar is based on an evolutionary improvement beginning with the XT3, Cray's third-generation massively parallel processing system, building on the T3D and T3E systems. Based on commodity AMD Opteron processors, most recently for instance the quad-core Barcelona system, a Cray custom interconnect, and a light-weight kernel (LWK) operating system, the XT3 was delivered in 2005. Each node consisted of an AMD Opteron model 150 (single core) processor, running at 2.4 GHz with 2 GBytes of DDR-400 memory. The nodes were connected by a SeaStar router through HyperTransport, in a 3-dimensional torus topology, and running the Catamount operating system. With 5,212 compute nodes, the peak performance of the XT3 was just over 25 TFLOPS [XT3a-c].

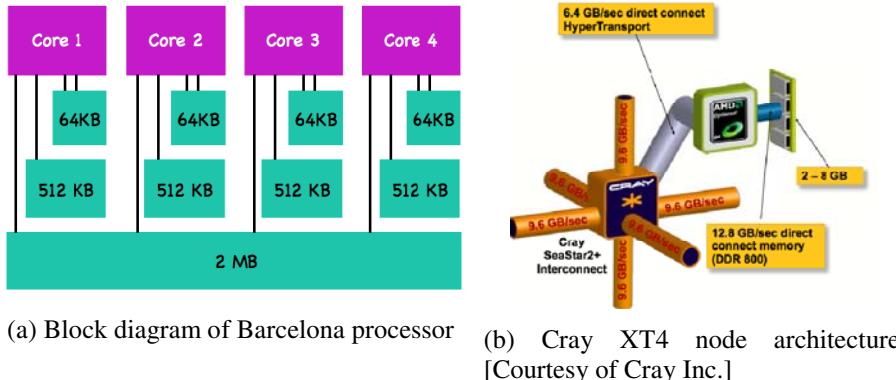


Fig. 1. Internals of a Quad-core based Cray XT4 node

Jaguar processors were upgraded to dual-core Opteron model 100 2.6 GHz processors in 2006, with memory per node doubled in order to maintain 2 GBytes per core. It was again upgraded April, 2007, with three major improvements: 6,296 nodes were added; memory on the new nodes was upgraded to DDR2-667, increasing memory bandwidth from 6.4 GBytes per second (GB/s) to 10.6 GB/s; and the SeaStar2 network chip connected the new nodes, increasing network injection bandwidth (of those nodes) from 2.2 GB/s to 4GB/s and increasing the sustained network performance from 4GB/s to 6GB/s. Thus with 23,016 processor cores, this so-called XT3/XT4 hybrid provided a peak performance of 119 TFLOPS [XT4a-b].

In spring 2008, Jaguar was again upgraded: 7,832 quad-core processors replace the 11,508 dual-core (illustrated in Figure 1, the interconnect is now fully SeaStar2, and the LWK is a customized version of Linux named Compute-Node Linux (CNL). Each compute node now contains a 2.1 GHz quad-core AMD Opteron processor and 8

GBytes of memory (maintaining the per core memory at 2 GBytes). As before, nodes are connected in a 3-dimensional torus topology, now with full SeaStar2 router through HyperTransport (see Figure 1(b)). This configuration provides 262 TFLOPS with 60 TBytes of memory.

3 Micro-benchmark and Application Details

3.1 HPCC Benchmark Suite

We used High Performance Computing Challenge (HPCC) benchmark suite to confirm micro-architectural characteristics of the system. HPCC benchmark suite [HPCCa-b] is composed of benchmarks measuring network performance, node-local performance, and global performance. Network performance is characterized by measuring the network latency and bandwidth for three communication patterns. The node local and global performance are characterized by considering four algorithm sets, which represent four combinations of minimal and maximal spatial and temporal locality: DGEMM/HPL for high temporal and spatial locality, FFT for high temporal and low spatial locality, Stream/Transpose (PTRANS) for low temporal and high spatial locality, and RandomAccess (RA) for low temporal and spatial locality. The performance of these four algorithm sets are measured in single/serial process mode (SP) in which only one processor is used, embarrassingly parallel mode (EP) in which all of the processors repeat the same computation in parallel without communicating, and global mode in which each processor provides a unique contribution to the overall computation requiring communication.

3.2 Application Case Studies

The application test cases are drawn from the workload configurations that are expected to scale to large number of cores and that are representative of Petascale problem configurations. These codes are large with complex performance characteristics and numerous production configurations that cannot be captured or characterized adequately in the current study. The intent is rather to provide a qualitative view of system performance using these test cases to highlight how the quad-core system upgrade has influenced the performance as compared to the preceding system configurations.

3.2.1 Fusion Application (AORSA)

The two- and three-dimensional All-ORDers Spectral Algorithm (AORSA [AORSA08]) code is a full-wave model for radio frequency heating of plasmas in fusion energy devices such as the International Thermonuclear Experimental Reactor5 (ITER) and the National Spherical Torus Experiment (NSTX) [Jaeger06-07]. AORSA operates on a spatial mesh, with the resulting set of linear equations solved for the Fourier coefficients. A Fast Fourier Transform algorithm converts the problem to a frequency space, resulting in a dense, complex- valued linear system. Parallelism is centered on the solution of the dense linear system, currently accomplished using a locally modified version of HPL [Dongarra90, Longau07]. Quasi-linear diffusion coefficients are then computed, which serve as an input to a separate application (Fokker-Plank solver) which models the longer term behavior of the plasma.

3.2.2 Turbulent Combustion Code (S3D)

Direct numerical simulation (DNS) of turbulent combustion provides fundamental insight into the coupling between fluid dynamics, chemistry, and molecular transport in reacting flows. S3D is a massively parallel DNS solver developed at Sandia National Laboratories. S3D solves the full compressible Navier-Stokes, total energy, species, and mass continuity equations coupled with detailed chemistry. It is based on a high-order accurate, non-dissipative numerical scheme and has been used extensively to investigate fundamental turbulent chemistry interactions in combustion problems including auto-ignition [Chen06], premixed flames [Sankaran07], and non-premixed flames [Hawkes07].

The governing equations are solved on a conventional three-dimensional structured Cartesian mesh. The code is parallelized using a three-dimensional domain decomposition and MPI communication. Spatial differentiation is achieved through eighth-order finite differences along with tenth-order filters to damp any spurious oscillations in the solution. The differentiation and filtering require nine and eleven point centered stencils, respectively. Ghost zones are constructed at the task boundaries by non-blocking MPI communication among nearest neighbors in the three-dimensional decomposition. Time advance is achieved through a six-stage, fourth-order explicit Runge-Kutta (R-K) method [Kennedy00].

4 Quantitative Evaluation and Analysis of Selected Features

4.1 Vector (SSE) Instructions

The Cray XT4 system is upgraded from dual-core Opteron to a single-chip, native quad-core processor called Barcelona. One of the main features of the quad-core system was quadrupling the floating point performance using a wider, 32-byte instruction fetch, and the floating-point units can execute 128-bit SSE operations in a single clock cycle (including the Supplemental SSE3 instructions Intel included in its Core-based Xeons). In addition, the Barcelona core has relatively higher bandwidth in order to accommodate higher throughput—internally between units on the chip, between the L1 and L2 caches, and between the L2 cache and the north bridge/memory controller.

In order to measure the impact of the new execution units with 128-bit vectorization support, we ran two HPCC benchmarks that represent scientific computation: DGEMM and FFT, in single processor (SP) where a single instance of an application runs on a single core and embarrassingly parallel (EP) where all cores execute an application without communicating with each other. We also measure performance per socket to estimate overall processor efficiencies. The quad-core XT4 has 4 cores per socket while the dual-core XT4 has two cores per socket. The results are shown in Figure 2. We observe a significant increase in per core performance for the dense-matrix computation benchmark (DGEMM), which is able to exploit the vector units. The FFT benchmarks on the other hand showed a modest increase in performance. Results in the EP mode when all four cores execute the same program revealed the impact of the shared L3 cache as the FFT performance slows down at a much higher rate for the quad-core system as compared to the dual-core and single-core XT platforms. The L3 behavior is detailed in the next section.

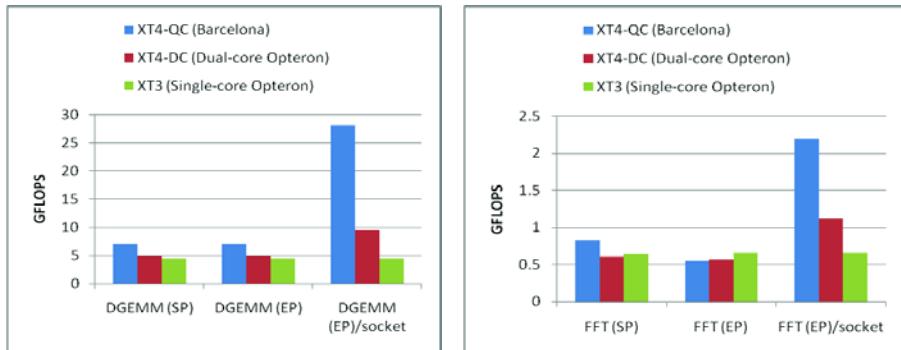


Fig. 2. HPCC DGEMM and FFT performance on different generations of Cray XT systems

Although there is a slowdown for FFT in the EP mode, we observe that per socket performance of the quad-core processor is significantly higher than that of the dual-core processor. We conclude that the significant performance boost per core brings in additional requirements for code development and generation for the quad-core processors. In other words, a misaligned and non-vector instruction could result in a code achieving less than a quarter of total achievable performance. Our two target applications highlighted the need for optimizing these vector operations.

AORSA, a fusion application, has a distinguished history running on the XT-series, allowing researchers to conduct experiments at resolutions previously unattainable [XT4a-c] executing at unprecedented computational scales. For example, the first simulations of mode conversion in ITER were run on the single-core XT3 [Jaeger06] on a 350 x 350 grid. On the dual-core XT3/XT4, this feat was again achieved, at increased resolution (500 x 500 grid), with the linear solver achieving 87.5 TFLOPS (74.8% of peak) on 22,500 cores [Jaeger07]. This same problem run on the quad-core XT increased this performance to 116.5 TFLOPS, and when run on 28,900 cores performance increased to 152.3 TFLOPS. Performance results for this scale are shown in Figure 3. Results are shown for the dual-core (DC) and quad-core (QC) processors with ScaLAPACK (Scal) and HPL (hpl) based solver. Moreover, experimental mixed-precision (mp) results are also shown in the figure. While impressive, relative to the theoretical peak performance has decreased from 74.8% to 61.6%. Although this is not unexpected due to the decreased clock speed and other issues associated with the increased number of cores per processor, we are pursuing further improvements. However, the time-to-solution (the relevant metric of interest) dropped from 73.2 minutes to 55.0 minutes, a decrease of 33%.

We expect performance of the solver phase to increase based on planned improvements to the BLAS library and the MPI implementation. In addition, we are experimenting with a mixed-precision approach [Langou07]. This capability is currently included in the Cray math library (-libsci) as part of the Iterative Refinement Toolkit (IRT). While this technique shows promise, it is not providing an improvement at the relevant problem scales. Although the condition of the matrix increases with resolution, this does not appear to be an issue. More likely is the use of the ScaLAPACK factorization routine within IRT compared with the HPL version: at 22,500 cores on the dual-core Jaguar, ScaLAPACK achieved 48 TFLOPS, whereas HPL achieved 87 TFLOPS.

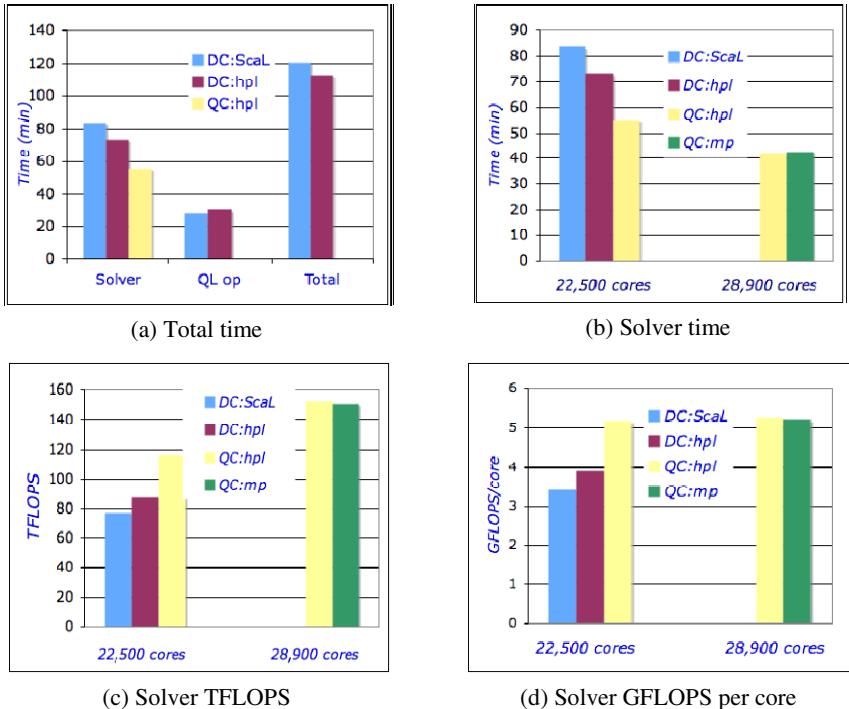


Fig. 3. AORSA performance on XT4 systems

The turbulent combustion application, S3D, is parallelized using a three-dimensional domain decomposition and MPI communication. Each MPI process is responsible for a piece of the three-dimensional domain. All MPI processes have the same number of grid points and the same computational load. Inter-processor communication is only between nearest neighbors in a logical three-dimensional topology. A ghost-zone is constructed at the processor boundaries by non-blocking MPI sends and receives among the nearest neighbors in the three-dimensional processor topology. Global communications are only required for monitoring and synchronization ahead of I/O. A comparison of dual-core and quad-core performance is shown in Table 1.

The initial port (Table 1) showed a decrease in performance, though less than that attributable to only the decrease in clock speed. This suggests that vectorization is occurring, though not as aggressively as desired. Special effort was applied to the computation of reaction rates, which consumed approximately 60% of overall runtime. Table 2 shows the effects of the compiler when able to vectorize code. Although for each category the number of operations increases, the proportion of operations occurring in vector mode increased by 233%, resulting in a decrease in runtime of this computation by over 20%.

Table 1. S3D single processor performance (weak scaling mode). The amount of work is constant for each process. “MPI mode” refers to the number of MPI processes and how they are assigned to each node: $-n$ is the total number of processes, $-N$ is the number of processes assigned to each quad-core processor node. Time is wall clock in units of seconds; “cost” is defined as micro-sec per grid point per time step. The “vec” columns show the performance after the code was reorganized for stronger vectorization.

Problem Size	MPI mode	Dual-core		Quad-core			
		Time	Cost	Time		Cost	
				vec	vec	vec	vec
$30 \times 30 \times 30$	$-n 1 -N 1$	404	150	415	333	154	123
$60 \times 30 \times 30$	$-n 2 -N 2$	465	172	430	349	159	129
$60 \times 60 \times 30$	$-n 4 -N 4$	n/a	n/a	503	422	186	156

Table 2. S3D reaction rate computation counters. (Values from dual-core Jaguar).

Counters	Before	After
	$\times 10^9$ operations	
Add	182	187
Multiply	204	210
Add + Mult	386	397
Load/Store	179	202
SSE	91	212

4.2 Deeper Memory Hierarchy (L3 Cache)

Another distinctive feature of the quad-core processor is the availability of an L3 cache that is shared among all four cores. There was no L3 cache in the predecessor Opteron processors. L3 serves as a victim cache for L2. L2 caches (not shared) are filled with victims from the L1 cache (not shared) i.e. after the L1 fills up rather than sending data to memory it sits in L2 for reuse. Hence, data-intensive applications could benefit from this L3 cache only if the working set is within cache range.

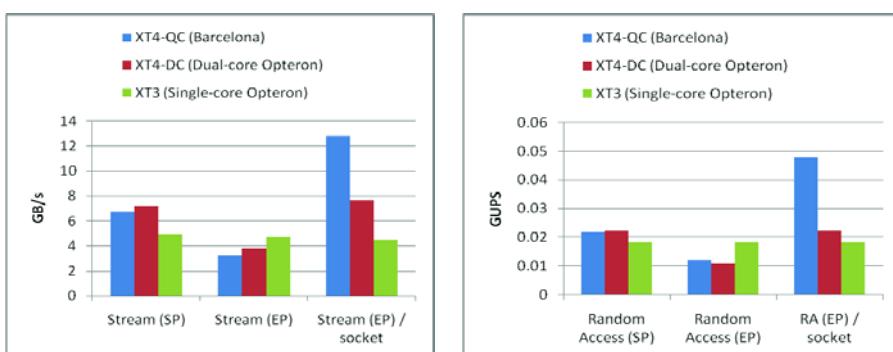


Fig. 4. HPCC memory performance benchmark results

Two HPCC memory performance benchmarks, stream and random access, were targeted to quantitatively evaluate performance of the memory sub-system. We compared the quad-core performance with the dual-core (XT4-DC) and single-core (XT3) AMD processors that preceded the latest quad-core (XT4-QC) processing nodes as shown in Figure 4. Both memory benchmarks highlight the effect of using a single core (SP mode) as compared to using all four cores simultaneously (EP mode) both for regular, single-strided (stream) access and random access benchmarks. Random memory access benchmarks highlight this cache behavior. We note that the shared resources in memory sub-system do account for slowdown in the EP mode, however this slowdown is less than by a factor of 4. In fact on the quad-core system, we have a relatively high per socket performance as compared to the dual-core system, which can be attributed to the shared L3 cache.

We have multiple applications that show slowdown in the quad-core or virtual node mode (VNM) modes as compared to single-core (SMP). In VNM mode 16 XT nodes are used while in SMP mode 64 XT nodes (256 cores) are reserved but only one core per node is used for 64 MPI tasks altogether. The S3D application has about a 25% slowdown in the mode where all four cores contribute to a calculation as compared to only single-core per processor. We collected hardware counter data using the PAPI library that confirms our findings [PAPI00]. L3 cache (shared between all four cores) behavior is measured and computed using the following PAPI native events. The L3 miss rate shows how frequently a miss occurs for a set of retired instructions. The L3 miss ratio indicates the portion of all L3 accesses that result in misses. Our results confirm that the L3 cache miss and request rate increase by a factor of two when using 4 cores per node versus using 1 core per node mode.

The most distinctive feature of the Petaflops XT5 system is the dual-socket, quad-core nodes as compared to a single-core socket node. In other words, there could be an additional level of memory and communication hierarchies that could be exposed to the application developers that are familiar with the quad-core XT4 memory sub-system. Although the optimization for the wide vector units would be beneficial for the XT5 system, the issues of memory sub-system are likely to become more complex since there will be 8 cores sharing the Hypertransport link on the XT5 node as compared to 4 cores on the XT4 node.

5 Conclusions and Future Plans

We have demonstrated how individual features of a system's hardware and software stack could influence performance of high-end applications on over a 250 Teraflops scale supercomputing platform. Our capability of comparing and contrasting performance and scaling of applications on multiple generations of Cray XT platforms, which share many system software and hardware features, enable us to not only identify the strategies to improve efficiencies on the current generation system but also prepare us to target the next-generation Petascale system. Since only a selection of features is studied in detail for this study, we plan on expanding the scope of this research by including application that have hybrid programming models to study the impact of within and across nodes and sockets. We are in process of working with application groups that have a flat MPI hierarchy models to explore and incorporate alternate work decomposition strategies on the XT5 platform.

Acknowledgements

This work was supported by the United States Department of Defense and used resources of the Extreme Scale Systems Center and the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the Department of Energy under Contract DE-ASC05-00OR22725.

References

- [Dagum98] Dagum, L., Menon, R.: OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science & Engineering* 5(1), 46–55 (1998)
- [Snir98] Snir, M., Gropp, W.D., et al. (eds.): *MPI – the complete reference* (2-volume set), 2nd edn. MIT Press, Cambridge (1998)
- [BGL05] Gara, A., et al.: Overview of the Blue Gene/L system architecture. *IBM Journal of Research and Development*, 49(2-3) (2005)
- [BGP08] Vetter, J.S., et al.: Early Evaluation of IBM BlueGene/P. In: *Proceedings of Supercomputing* (2008)
- [XT3a] Camp, W.J., Tomkins, J.L.: Thor's hammer: The first version of the Red Storm MPP architecture. In: *Proceedings of Conference on High Performance Networking and Computing*, Baltimore, MD (November 2002)
- [XT3b] Vetter, J.S., Alam, S.R., et al.: Early Evaluation of the Cray XT3. In: *Proc. IEEE International Parallel and Distributed Processing Symposium, IPDPS* (2006)
- [XT4a] Alam, S.R., Barrett, R.F., et al.: Cray XT4: An Early Evaluation for Petascale Scientific Simulation. In: *Proceedings of the IEEE/ACM Conference on Supercomputing SC 2007* (2007)
- [XT4b] Alam, S.R., Barrett, R.F., et al.: The Cray XT4 Quad-core: A First Look. In: *Proceedings of the 50th Cray User Group* (2008)
- [Kelly05] Kelly, S., Brightwell, R.: Software architecture of the lightweight kernel, catamount. In: *Proceedings of the 47th Cray User Group* (2005)
- [HPCCa] Luszczek, P., Dongarra, J., et al.: Introduction to the HPC Challenge Benchmark Suite (March 2005)
- [HPCCb] High Performance Computing Challenge Benchmark Suite Website, <http://icl.cs.utk.edu/hpcc/>
- [AORSA08] Barrett, R.F., Chan, T., et al.: A complex-variables version of high performance computing LINPACK benchmark, HPL (2008) (in preparation)
- [Jaeger06] Jaeger, E.F., Berry, L.A., et al.: Self-consistent full-wave and Fokker-Planck calculations for ion cyclotron heating in non-Maxwellian plasmas. *Physics of Plasmas* (May 13, 2006)
- [Jaeger07] Jaeger, E.F., Berry, L.A., et al.: Simulation of high power ICRF wave heating in the ITER burning plasma. In: Jaeger, E.F., Berry, L.A. (eds.) *Proceedings of the 49th Annual Meeting of the Division of Plasma Physics of the American Physical Society*, vol. 52. *Bulletin of the American Physical Society* (2007)
- [Dongarra90] Dongarra, J.J., DuCroz, J., et al.: A set of level 3 basic linear algebra subprograms. *ACM Trans.on Math. Soft.* 16, 1–17 (1990)
- [Langou07] Langou, J., Luszczek, P., et al.: Tools and techniques for exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems). In: *Proc. ACM/IEEE Supercomputing (SC 2006)* (2006)

- [Chen06] Chen, J.H., Hawkes, E.R., et al.: Direct numerical simulation of ignition front propagation in a constant volume with temperature inhomogeneities I. fundamental analysis and diagnostics. *Combustion and flame* 145, 128–144 (2006)
- [Sankaran07] Sankaran, R., Hawkes, E.R., et al.: Structure of a spatially developing turbulent lean methane-air Bunsen flame. *Proceedings of the combustion institute* 31, 1291–1298 (2007)
- [Hawkes07] Hawkes, E.R., Sankaran, R., et al.: Scalar mixing in direct numerical simulations of temporally evolving nonpremixed plane jet flames with skeletal CO-H₂ kinetics. *Proceedings of the combustion institute* 31, 1633–1640 (2007)
- [Kennedy00] Kennedy, C.A., Carpenter, M.H., Lewis, R.M.: Low-storage explicit Runge-Kutta schemes for the compressible Navier-Stokes equations. *Applied numerical mathematics* 35(3), 177–264 (2000)
- [Scal] The ScaLAPACK Project, <http://www.netlib.org/scalapack/>
- [Petit04] Petitet, A., Whaley, R.C., Dongarra, J.J., Cleary, A.: HPL: A portable high-performance LINPACK benchmark for distributed-memory computers (January 2004), <http://www.netlib.org/benchmark/hpl>
- [PAPI00] Browne, S., Dongarra, J., et al.: A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters. In: *Proceedings of Supercomputing* (2000)

Topic 5

Parallel and Distributed Databases

Introduction

Alex Szalay*, Djoerd Hiemstra*, Alfons Kemper*, and Manuel Prieto*

Euro-Par Topic 5 addresses data management issues in parallel and distributed computing. Advances in data management (storage, access, querying, retrieval, mining) are inherent to current and future information systems. Today, accessing large volumes of information is a reality: Data-intensive applications enable huge user communities to transparently access multiple pre-existing autonomous, distributed and heterogeneous resources (data, documents, images, services, etc.). Data management solutions need efficient techniques for exploiting and mining large datasets available in clusters, peer to peer and Grid architectures. Parallel and distributed file systems, databases, data warehouses, and digital libraries are a key element for achieving scalable, efficient systems that will cost-effectively manage and extract data from huge amounts of highly distributed and heterogeneous digital data repositories.

Each paper submitted to Euro-Par's topic Parallel and Distributed Databases was reviewed by at least three reviewers. Of 11 papers submitted to the topic this year, 3 were accepted, which makes an acceptance rate of 27 %. The three accepted papers discuss diverse issues: database transactions, efficient and reliable structured peer-to-peer systems, and selective replicated declustering.

In their paper *Unifying Memory and Database Transactions*, Ricardo Dias, and João Lourenço present a simple but powerful idea: Combining software transactional memory with database transactions. The paper proposes to provide unified memory and database transactions by integrating the database transaction control with a software framework for transactional memory. Experimental results show that the overhead for unified transactions is low. It is likely that the approach lowers the burden on the application developer.

The paper by Hao Gong, Guangyu Shi, Jian Chen, and Lingyuan Fan, *A DHT Key-Value Storage System with Carrier Grade Performance*, tries to achieves reliability and efficiency in peer-to-peer systems in order to support Telecom services. The proposed design is based on: Adopting a two-layer distributed hash table, embedding location information into peer IDs; Providing one-hop routing by enhancing each peer with an additional one-hop routing table, where super-peers are in charge of updating and synchronizing this routing information. Finally, the approach replicates subscriber data on multiple peers.

Finally, Kerim Oktay, Ata Turk, and Cevdet Aykanat present a paper on *Selective Replicated Declustering for Arbitrary Queries*. The authors present a new

* Topic Chairs.

algorithm for selective replicated declustering for arbitrary queries. The algorithm makes use of query information available in order to decide on the data assignment to different disks and on which data to replicate respecting space constraints. Further, it is described how to apply the proposed algorithm in a recursive way for obtaining a multi-way replicated declustering. Experiments show the algorithm outperforms existing replicated declustering schemes, especially for low replication constraints.

We would like to thank the external reviewers for this Euro-Par topic for their helpful reviews, as well as the Euro-Par 2009 organizing committee for excellent coordination of the work.

Unifying Memory and Database Transactions

Ricardo J. Dias and João M. Lourenço

CITI—Centre for Informatics and Information Technology, and
Departamento de Informática, Universidade Nova de Lisboa
Portugal
`{rjfd,joao.lourenco}@di.fct.unl.pt`

Abstract. Software Transactional Memory is a concurrency control technique gaining increasing popularity, as it provides high-level concurrency control constructs and eases the development of highly multi-threaded applications. But this easiness comes at the expense of restricting the operations that can be executed within a memory transaction, and operations such as terminal and file I/O are either not allowed or incur in serious performance penalties. Database I/O is another example of operations that usually are not allowed within a memory transaction. This paper proposes to combine memory and database transactions in a single unified model, benefiting from the ACID properties of the database transactions and from the speed of main memory data processing. The new unified model covers, without differentiating, both memory and database operations. Thus, the users are allowed to freely intertwine memory and database accesses within the same transaction, knowing that the memory and database contents will always remain consistent and that the transaction will atomically abort or commit the operations in both memory and database. This approach allows to increase the granularity of the in-memory atomic actions and hence, simplifies the reasoning about them.

1 Introduction

Software transactional memory (STM) is a promising concurrency control approach to multithreaded programming. More than a concurrency control mechanism, it is a new programming model that brings the concept of transactions into the programming languages, by way of new language constructs or as a simple API and supporting library. Transactions are widely known as a technique that ensure the four ACID properties : Atomicity (A), Consistency (C), Isolation (I) and Durability (D).

Memory transactions, with roots in the database transactions, must only ensure two of the ACID properties: Atomicity and Isolation. The Consistency and Durability properties may be dropped, as memory transactions operate in volatile memory (RAM). Volatile memory does not have persistence properties and does not have a fixed logical structure, like a database system, over which one can make consistency assertions.

In the past few years, several STM frameworks have been developed. Most of these STM frameworks take the form of software libraries providing an API to support the transactional model to the application [2,3,4,5]. This library-based approach allows the rapid prototyping of algorithms and their performance evaluation. Some other STM frameworks extend existing programming languages with transactional constructs supported directly by the compiler [6,7,8,9]. Most of these frameworks focus in managed languages such as Java, C#, and Haskell, while some other target unmanaged languages like C and C++.

One drawback of using STM lies in the execution of partially- and non-transactional operations within a transaction [10]. Pure-transactional operations are undone automatically by the STM transactional framework when a transaction aborts, e.g., changing the contents of memory. Non-transactional operations simply cannot be undone, e.g., writing data to the terminal. Partially-transactional operations are either revertible or compensable, and can be undone at some expense, e.g., explicit memory management operations and I/O to disk files. Some STM frameworks opt to not allow the execution of partially- and non-transactional operations within memory transactions [7]. Some others force the memory transactions to execute non- and partially-transactional operations in mutual exclusion to all other transactions in the system [9,11].

Another example of partially-transactional (revertible) operations are accesses to a transactional database from within a memory transaction. Particularly interesting is the case where the application decides to commit the database transaction within a memory transaction. If the memory transaction later needs to abort, it would be necessary to rollback the already committed database transaction. This problem has been briefly enunciated in the past [12], but to our best knowledge to date no other work has addressed this matter.

This paper proposes a solution to the above problem by widening the transactional model scope to cover, without differentiating, both memory and database operations. Thus, the users are allowed to freely intertwine memory and database accesses within the same transaction, knowing that the memory and database contents will always be consistent and that the transaction will either abort or commit the operations in both memory and database atomically.

The remaining of this paper is organized as follows. Section 2 defines the unified model and its properties. Section 3 describes the implementation of the unified model using a specific STM framework and database management system. Section 4 evaluates the performance of our implementation of the unified model and compares it with other alternative approaches and, finally, Sect. 5 closes with some concluding remarks.

2 The Unified Model

Working with two or more transactional models in the same application, each with its own set of properties and guarantees, may become unbearable. This paper proposes an unified transactional model, merging two currently popular transactional models: memory and database transactions.

Transactions in the unified model can enclose two main classes of operations, memory and database operations. In the first class, two types of operations are allowed, read and write of memory locations. In the second class, all transactional operations supported by database are allowed.

The unified model preserves the minimum common set of properties from both of the transactional models it unifies—Atomicity and Isolation—, allowing to define sets of operations as transactions, acting upon the memory and the database atomically and isolated from other concurrent transactions. The unified model also inherits the Consistency and Durability properties from the database management system, but these properties only apply to database operations.

The atomicity and isolation properties apply to any operation valid in the unified model. This includes both memory and database transactional operations. A transaction will not see intermediate memory nor database states of other transactions, and all the effects caused in both memory and database will either persist upon the transaction commit or be rolled back in case of an abort.

The database transactional model allows multiple applications to access the database concurrently and the ACID properties always hold. The transactional memory model, however, only applies to the memory shared between multiple control flows, typically multiple threads within a single application. The unified model will hold the most restrictive properties, i.e., will hold the AI properties for a single multithreaded application. This assumption can be somewhat relaxed when using a distributed transactional memory framework [13]. In this case the application breaks the physical node barrier, but still has to be a single distributed application whose multiple components are cooperating using the distributed STM framework. In other words, the unified model assumes exclusive access to the database, i.e., that no other application is accessing the database at the same time.

3 Implementation

The unified model was implemented resorting to a library based STM framework, the Consistent Transactional Layer [14] (CTL), a STM implementation for the C programming language and derived from the TL2 [2] library. CTL extends the TL2 framework with a large set of new features and optimizations [15]. CTL also implements a full featured handler system that allow the users to define reverting operations. These operations are executed in key moments of a memory transaction and allow to revert the effects of partially-transactional operations executed within a memory transaction.

3.1 CTL Handler System

The CTL handler system [10] extends the life-cycle of memory transaction by executing user-defined functions in specific key moments. Users can define five types of handlers which are executed by the CTL run-time at four different key moments of the transaction life-cycle: *prepare-commit* and *pre-commit* handlers

are executed before the transaction commits; *pos-commit* handlers are executed after the transaction commits; *pre-abort* handlers are executed before a transaction aborts; and *pos-abort* handlers are executed after the transaction aborts.

Both prepare-commit and pre-commit handlers are executed after validating the memory transactions and, thus, may assume that the memory transaction will commit. The former may still force the transaction to abort while the latter cannot do so and must definitely assume that the transaction will commit. The adequate combination of prepare-commit and pre-commit handlers allows to execute a *two-phase-commit protocol* [16] between several transactional subsystems in which one of them is the memory transaction. The two-phase-commit protocol includes two main phases: the preparation phase, where all transactions must first agree whether they will commit or abort; and the commit/abort phase, where all the transactions must perform the decision previously agreed. If the decision was to commit, the transaction manager will request all the transactions to commit, otherwise it will request all the transactions to abort.

3.2 Unified Model Implementation

The unified model was implemented using two distinct transactional systems: one for memory transactions and another for database transactions. Dealing with two autonomous transactional systems as a single one requires that the commit/abort phase of both transactional systems to be atomic: both must either commit or abort. The *two-phase-commit* protocol (2PC) allows to commit N transactions, from N different transactional systems, atomically. This algorithm was implemented by having the STM framework playing two roles, one as the 2PC controller, and another as a partner in the 2PC protocol together with the database system.

The database is accessed by way of an ODBC interface for the C programming language. This approach allows to use any ODBC compliant database management system and initial experiments were made with two different ODBC compliant databases, DB2 and PostgreSQL. This dual database tests were essential to functionally validate the approach, but experiments demonstrated that PostgreSQL outperformed DB2 to the point that it became irrelevant to run performance tests with DB2. Thus, this paper only reports on performance tests with PostgreSQL.

Without support for the database prepare phase, the commit of the database transaction will be attempted as a prepare-commit handler. When the prepare-commit handlers are executed, the memory transaction has already been validated, thus it is known that the memory transaction will not abort due to concurrency conflicts. As prepare-commit handlers can still abort the transaction, this approach is safe as long as the database commit is the last prepare-handler to be executed, and this rule can be enforced by the TM framework. If the database commit is not successful, hence the database transaction has aborted, then the last of the prepare-commit handlers will also fail and the memory transaction will be rolled back. In the presence of a concurrency conflict, either in memory or in the database, the memory

transaction will abort, the pre-abort handler will be executed and the database transaction will also be aborted.

To ease the work of registering all the necessary handlers to allow the execution of the 2PC protocol with the transactional memory framework acting as the controller, a new call, `TxDBStart`, was introduced into the CTL transactional memory API to replace the call previously used to start a new memory transaction. Figure 1 shows the definition of this new front-end.

```

1 void TxDBStart (Thread *Self, HDBC dbc, int roflag) {
2     TxStart (Self, roflag);
3     _ctl_register_prepare_handler (Self, do_commit, (void *)dbc);
4     _ctl_register_pre_abort_handler (Self, do_abort, (void *)dbc);
5 }
```

Fig. 1. `TxStart` front-end for using database transactions

The first and third parameters of `TxDBStart` are the same as for the CTL `TxStart` function. The second parameter is the database connection handler for the ODBC interface. Thus, obtaining the ODBC connection handler to the database and replacing all the calls to `TxStart` to the new `TxDBStart` is all that is needed for an application to switch from the pure transactional memory model into the new unified model.

`TxDBStart` starts by calling `TxStart` and initiating a new memory transaction, then it registers the `do_commit` function as a *prepare-commit handler* and the `do_abort` function as a *pre-abort handler*. Thus, when the memory transaction terminates either by committing or aborting, the appropriate handler will be executed and the database transaction will also terminate. The definitions of `do_commit` and `do_abort` are similar, as both only call the ODBC `SQLEndTran` function with the appropriate flag indicating whether to commit or abort.

Figure 2 shows the implementations of the `do_commit` function. The database transaction is implicitly started by the ODBC upon the first operation over the database and will end upon the commit or abort of the transaction.

The database connection handler is passed to the function `TxDBStart`, identifying which database should be used in the current transaction. It is possible to use different databases in different transactions, but it is not possible to use more than one database per transaction in this implementation of the unified model. This limitation is due to the fact that the ODBC does not support

```

1 int do_commit (Thread *Self, void *args) {
2     SQLRETURN ret;
3     ret = SQLEndTran (SQL_HANDLE_DBC, (HDBC)args, SQL_COMMIT);
4     return (SQL_SUCCEEDED(ret) != 0);
5 }
```

Fig. 2. `do_commit` function handler definition

the prepare service in the database. Transactions that are known to not access the database can still use the original `TxStart` instead of the newly defined `TxDBStart`.

Guaranteeing that both memory and database transactions follow the 2PC protocol and commit and abort atomically, is not sufficient to guarantee the isolation property for the unified model. Some stronger requirements concerning the order in which concurrent transactions are scheduled must be fulfilled.

3.3 Implementation Requirements

Most transactional database systems allow to relax the isolation level, by admitting the execution of non-serializable transaction schedules [17], in order to increase the throughput of transaction processing. To guarantee the provision of Isolation to the unified model, both transactional models must run under the same isolation level which must be the strongest (most restrictive) from both. Since the common isolation level for transactional memory is full serialization, the database transactional system used must also run in full serialization isolation level. Another requirement for the correctness of the unified model is that both transactional systems generate equal serialization schedules. Serialization schedules are created dynamically and independently by both memory and database transactional systems, and the generated schedules are affected by the isolation level and by the concurrency control policies.

Although the model as described in Sect. 2 is generic enough to unify memory and database transactional models, the transactional memory framework and the database management system must be carefully chosen and parametrized to satisfy the two requirements described above.

3.4 Prototype Evaluation

We implemented a prototype using the PostgreSQL database management system [18]. Although PostgreSQL supports a Serializable Isolation Level, its performance is much worse than Snapshot Isolation Level (SIL). However, using this DBMS in SIL with no other modifications would not satisfy the requirements to support Isolation in the unified model. The problem with SIL is that the transactional system does not detect conflicts between two transactions where one is reading data from a record and the other is writing data into the same record. Once this type of conflicts are detected, there is no need to impose stronger restrictions on the isolation level.

One possible approach to force the detection of the read-write conflicts was to force read operations to be treated as write operations by the DBMS [19]. A simple way to achieve such a goal (with limitations) would be to force, for each `SELECT` statement of a data item D_i , to update the value of the data item to itself, i.e., force the operation $V(D_i) = V(D_i)$, followed by the desired `SELECT` statement. This solution would transform all read operations into write operations and would therefore impose a strong performance overhead. The PostgreSQL database [18] implements two variants of the `SELECT` SQL statement to address

such problem, the statements `SELECT FOR UPDATE` and `SELECT FOR SHARE`. The difference between the two statements is that `SELECT FOR UPDATE` acquires an exclusive lock of the rows being accessed while `SELECT FOR SHARE` acquires a shared lock, thus permitting other `SELECT FOR SHARE` statements to execute concurrently over the same rows. If any row is locked by an exclusive or shared lock, any write attempt to the locked row will block.

With this solution, the transaction schedules generated by the DBMS and by the transactional memory framework were identical, thus satisfying the requirements for the Isolation property to hold.

4 Evaluation of the Unified Model

The evaluation of the prototype of the unified model was twofold: functional and performance. Functional evaluation aimed at verifying the correctness of the system behavior, even under very stressing conditions, and its fitness to application development. Performance evaluation aimed at comparing absolute performance and scalability of our approach with coarse and finer grain locks.

The testing application stores and retrieves scientific articles from a database. The articles can be indexed by author name, by keywords, or both. The article repository has an interface with four services: insert an article, remove an article, find an article by author, and find an article by keyword. Depending on the version of the testing application, each service will access multiple shared data structures in main memory (when applicable), multiple tables in the database, or both. The benefits of this approach depend on the application being able to store part of the relevant information in main memory, thus avoiding to access the database for read-only operations.

The application is divided in two components: a server and a client. The server will manage the repository, allowing concurrent calls to the repository interface. The client is a single threaded component issuing a sequence of service requests to the server. This application will use the unified model to maintain an in-memory replica of the database indexation structures. As the unified model guarantees the consistency between both data repositories, if the application is closed all information still persists in database and when the application is restarted all indexation information is reloaded into main memory.

4.1 Database Model

The database scheme is very simple. It has three entities: articles, authors, and keywords. An article is represented by an internal *id*, a title, and the file path to the article document. An author is represented by its name (we assume that each author name is unique). A keyword is represented by itself (and must also be unique among the keywords). Each article must have at least one author and one keyword, but may have more. Figure 3 illustrates the entity-relation model of the database scheme just described.



Fig. 3. Application database entity-relation model

The relation between authors and articles is supported by an association table where each row makes the link between an author and an article. The same applies to the relation between keywords and articles.

4.2 Memory Data Structures

The indexing structures in memory are represented by a single linked list and a hash table. The structure is simple: one hash table for indexing authors and another for indexing keywords. Each element of the hash table is a list of article identifiers that are associated with that author or keyword.

The single linked list was implemented using CTL to protect it from concurrent accesses. This list implements three operations: insert an element, remove an element, and lookup for an element. Each of these operations uses the handler system, described in Sec 3.1, to manage the memory allocation and deallocation of list nodes inside memory transactions.

The hash table was also implemented using CTL to protect it from concurrent accesses. It also implements three operations: insert an element, remove an element, and lookup for an element. Similar to the linked list, each hash table operation also resorts to the handler system to manage memory allocation and deallocation inside memory transactions.

4.3 Description of Repository Operations

Inserting an article into the repository requires several steps to be executed in sequence. First the article is inserted into the database. If the database insertion succeeds (meaning that the article was not yet in the database), then the associations between the article and each of its authors are also inserted into the database. References to this article are also inserted, one for each author, in the hash table that represents the association in main memory. A similar process is executed for the keywords. Removing an article also requires several steps. First, the article is removed from the memory hash tables that maps authors to articles. Then, a similar operation is executed to remove that same association from the database. A similar process is executed for the keywords. Finding an article in the repository by author or by keyword involves only indexing the respective hash table and, for every article in the list, retrieve the info from the database.

The operations of insert and remove from the database repository, only use **INSERT** and **DELETE** SQL statements and do not use any **SELECT** statement. In this case, where we only perform write operations in database, no read-write conflict will ever occur, and therefore we can use the PostgreSQL DBMS resorting to the standard SQL **SELECT** statement.

4.4 Functional Evaluation

To validate the correction of the unified model algorithm, we developed an alternative version of the same testing application that would periodically validate the coherence between the data present in memory against its equivalent in the database. The assertion test was: *every data that is in memory must be also in database*. An assertion failure would mean that a coherency problem was found, either as a bug in the algorithm or in its implementation. We made several long runs of the application. In each run, the application was periodically paused to verify the consistency assertion, and later resumed. Although this test only provides statistical confidence on the correction of our algorithm, we must say that in the many accumulated hours of execution, the assertion was never broken.

4.5 Performance Evaluation

The performance of the unified model was evaluated by measuring the transactional throughput (completed transactions per time unit). A total of four versions of the application have been developed: one using the unified model as described above; another using coarse grain locks, where each repository operation was protected with a global lock; a third using finer grain locks, where each hash table bucket has a separate lock; and finally a version where the indexing structure was not replicated in main memory, i.e., the lookup operation required querying the database with `SELECT` statements.

Since each repository operation deals with more than one hash table, the implementation for finer-grain locks was very complex and error prone, as any small mistake in the management of the locks, including the order in which they were acquired and released, would cause a deadlock.

Each of the application variants were tested in four different environments: read dominant context and a write dominant contexts; and low and high contention contexts. The tests are characterized by three type of operations: `insert`, `remove`, and `lookup`. The insert and remove operations are *read-write* operations,

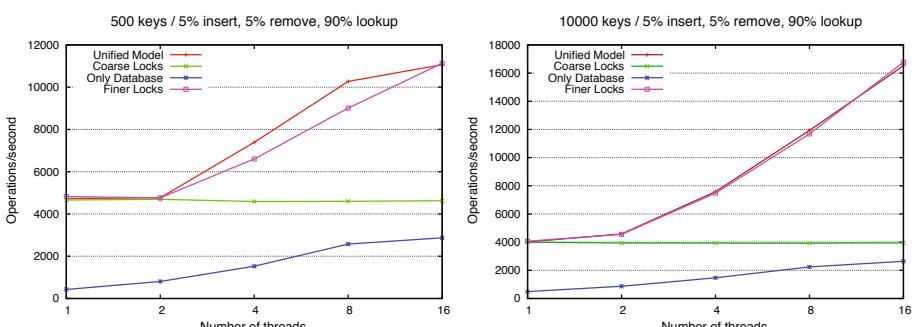


Fig. 4. Article repository used in a read dominant context, with high contention (left) and low contention (right)

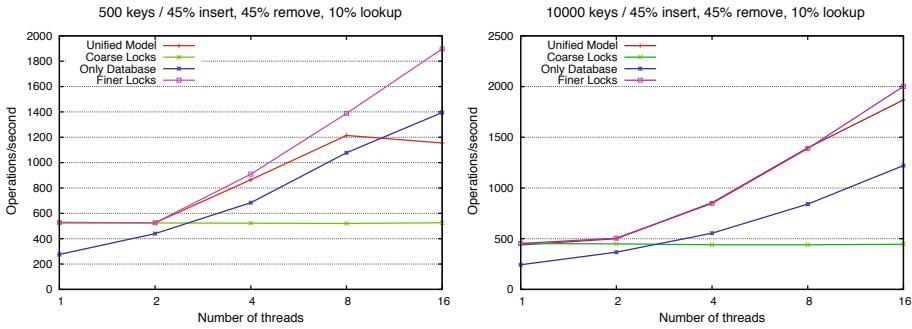


Fig. 5. Article repository used in a write dominant context, with high contention (left) and low contention (right)

while the lookup by author or by keyword operations are *read-only*. Each operation has a predefined probability defined as an application parameter. The maximum number of different articles to be inserted in the repository will control the contention level and will be also defined as an application parameter.

The tests were performed in a Sun Fire X4600 M2 x64 server, with eight dual-core AMD Opteron Model 8220 processors @ 2.8 GHz, 1024 KBytes of cache in each processor and a total of 32 GByte of RAM. The database management system used was PostgreSQL 8.3. Figures 4 and 5 show the results for the different testing configurations. In these tests, each article always had two authors and two keywords associated.

The first remark is that in read-dominant contexts (Fig. 4) the database version performance is poor comparing to the other versions. This was an expected result because the database has to access the secondary memory for every read, while the other versions only have to access the main memory which is a few orders of magnitude faster. Another important aspect that can be identified from the graphics is that the difference between finer grain locks and our unified model is solely affected by the contention level. Our unified model scales as well as finer grain locks for low level contention environments, and development of the application version using the unified model implementation is much simpler than the equivalent version with finer grain locks. The scalability of the unified model decreases when the contention level increases. This is more notorious when the number of parallel threads (each running a transaction) reach the maximum numbers of processors available. We believe this is due to the CPU time wasted by transactions that abort by contention conflicts, while the finer grain locks version blocks the thread when facing contention, never wasting CPU time.

For write-dominant contexts (Fig. 5), the database version scales almost as well as the other ones. This is due to the fact that now almost all the operations (90%) are either inserts or removes, and in both cases it is mandatory to always access the database. Please note that the vertical scales in the read- and write-dominant contexts are different, and that here is no performance improvement in the database-only version for the write-dominant context. In this case, all the

remaining versions that use transactional memory are performing much worse, as the vast majority of the operations require the execution of database updates.

5 Concluding Remarks

This work has proposed a new approach to unify the memory and database transactional models as a single one. The proposed model still differentiates between memory and database operations, but allow them to be freely intertwined. The unified model guarantees that the Atomicity and Isolation properties hold for both memory and database operations. Additionally, it guarantees that memory and database contents are consistent if changed within the same transaction. Although the unified model was implemented using a specific transactional memory framework and a specific DBMS, we presented the implementation requirements for achieving the same result with other transactional systems.

The unified model is a high level approach to concurrency management covering both memory and database operations. It is considerably faster than database-only solutions, and much simpler to use than those based in finer grain lock. In the future we will extend the unified model to support transaction nesting, will evaluate it with standard benchmarks such as TPC-C and TPC-W, and compare it with hybrid in-memory/on-disk databases.

Acknowledgements

Our thanks to the reviewers of this paper for their valuable comments. This work was partially supported by Sun Microsystems and Sun Microsystems Portugal under the “Sun Worldwide Marketing Loaner Agreement #11497”, by the Centro de Informática e Tecnologias da Informação (CITI) and by the Fundação para a Ciência e Tecnologia (FCT/MCTES) in the Byzantium research project PTDC/EIA/74325/2006 and research grant SFRH/BD/41765/2007.

References

1. Gray, J., Reuter, A.: *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco (1992)
2. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)
3. Cachopo, J., Rito-Silva, A.: Versioned boxes as the basis for memory transactions. *Sci. Comput. Program.* 63, 172–185 (2006)
4. Herlihy, M., Luchangco, V., Moir, M., William, N., Scherer, I.: Software transactional memory for dynamic-sized data structures. In: PODC 2003: Proceedings of the twenty-second annual symposium on Principles of distributed computing, pp. 92–101. ACM, New York (2003)
5. Luke, D., Marathe, V.J., Spear, M.F., Scott, M.L.: Capabilities and limitations of library-based software transactional memory in c++. In: Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing, Portland, OR (2007)

6. Harris, T., Fraser, K.: Language support for lightweight transactions. In: OOPSLA 2003: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp. 388–402. ACM, New York (2003)
7. Harris, T., Marlow, S., Peyton-Jones, S., Herlihy, M.: Composable memory transactions. In: PPoPP 2005: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 48–60. ACM, New York (2005)
8. Felber, P., Fetzer, C., Müller, U., Riegel, T., Süßkraut, M., Sturzrehm, H.: Transactionalizing applications using an open compiler framework. In: Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing (2007)
9. Ni, Y., Welc, A., Adl-Tabatabai, A.R., Bach, M., Berkowits, S., Cowrie, J., Geva, R., Kozhukow, S., Narayanaswamy, R., Olivier, J., Preis, S., Saha, B., Tal, A., Tian, X.: Design and implementation of transactional constructs for c/c++. SIGPLAN Not. 43, 195–212 (2008)
10. Dias, R., Lourenço, J., Cunha, G.: Developing libraries using software transactional memory. ComSIS 5, 104–117 (2008)
11. Blundell, C., Lewis, E.C., Martin, M.M.K.: Unrestricted transactional memory: Supporting i/o and system calls within transactions. Technical Report CIS-06-09, Department of Computer and Information Science, University of Pennsylvania (2006)
12. Harris, T.: Exceptions and side-effects in atomic blocks. Sci. Comput. Program. 58, 325–343 (2005)
13. Kotselidis, C., Ansari, M., Jarvis, K., Luján, M., Kirkham, C., Watson, I.: Distm: A software transactional memory framework for clusters. In: ICPP 2008: Proceedings of the 2008 37th International Conference on Parallel Processing, Washington, DC, USA, pp. 51–58. IEEE Computer Society, Los Alamitos (2008)
14. Cunha, G.: Consistent state software transactional memory. Master's thesis, Universidade Nova de Lisboa (2007)
15. Lourenço, J., Cunha, G.: Testing patterns for software transactional memory engines. In: PADTAD 2007: Proceedings of the 2007 ACM workshop on Parallel and distributed systems: testing and debugging, pp. 36–42. ACM, New York (2007)
16. Gray, J.: Notes on data base operating systems. Operating Systems, 393–481 (1978)
17. Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A critique of ansi sql isolation levels. In: SIGMOD 1995: Proceedings of the 1995 ACM SIGMOD international conference on Management of data, pp. 1–10. ACM, New York (1995)
18. Postgresql database management system, <http://www.postgresql.com>
19. Fekete, A., Liarokapis, D., O'Neil, E., O'Neil, P., Shasha, D.: Making snapshot isolation serializable. ACM Trans. Database Syst. 30, 492–528 (2005)

A DHT Key-Value Storage System with Carrier Grade Performance

Guangyu Shi¹, Jian Chen¹, Hao Gong¹, Lingyuan Fan¹, Haiqiang Xue²,
Qingming Lu¹, and Liang Liang¹

¹ Huawei Technologies Co., Ltd
Shenzhen, 518129, China

{shiguangyu, jchen, haogong}@huawei.com

² China Mobile Communications Co.
Beijing, 100053, China
xuehaiqiang@chinamobile.com

Abstract. The Peer-to-Peer (P2P) technology being widely adopted in today's both academic research and practical service providing, has many potential advantages and achieves a great success in Information Technology scope. Recently some researchers have proposed that P2P inspired architecture also might be one choice for the telecom network evolution. Most of such works adopted structured P2P (DHT) as the basic solutions, but they seldom discussed how to eliminate the huge gap between the telecom underlay performance requirement and the performance of existed DHT which mainly originated from the Internet applications. This paper presents the design and implementation of SandStone, a DHT based key-value storage system with carrier grade performance, such as good scalability, strong consistency and high reliability, which could be deployed as the cornerstone in such new P2P inspired networking architectures.

Keywords: Peer-to-Peer, DHT, Telecom Network, Key-Value Storage.

1 Introduction

Reliability and efficiency at massive scale is one of the important challenges in telecom applications, even the slightest outage or congestion has significant financial consequences and impacts customer experience. Although most of the current telecom networks were strictly central server based, many researchers have proposed a new networking architecture inspired by the P2P paradigm, in order to benefit from the advantages of decentralization such as high scalability and cost-effectiveness.

In the last few years, a number of systems and prototypes have been proposed to incorporate P2P technology into the field of telecommunications. For example, P2PSIP [1] proposed a DHT based conversational signaling system. Reference [2] and [3] presented two kinds of distributed IP Multimedia Subsystem (IMS) architectures, utilizing the DHT as the basic register/lookup functional entity, contrasted with central servers based on present IMS HSS/HLR. It's evident that methodology behind these works is to push P2P into the telecom underlay application, although P2P technology emerged as an Internet overlay technology.

On the other hand, P2P Distributed Storage Systems have been developed by leaps and bounds in recent years; typical examples include GFS [4], OceanStore [5], Pond [6], BitVault [7], Ceph [8], Dynamo [9], and WheelFS [10]. In the principles of these systems, the authors have a unified view that structured P2P or DHT should be used as the basic key-value storage system infrastructure. Nevertheless, in comparison with the Internet overlay applications, apparently telecom applications have a much stricter performance requirement for real-time response, reliability and consistency. Unfortunately, so far as we know, none of them has seriously considered the question that how to enhance DHT technology to face such performance requirements challenges.

In this paper, we present the design and implementation of SandStone, a highly decentralized, loosely coupled, service oriented storage system architecture consisting of thousands of PC peers. The name “SandStone” stands for transforming the enormous sand-like tiny and common PCs’ capability into an invincible infrastructure cornerstone of distributed storage systems, to provide a “Carrier Grade Storage” experience. It means, the system is tested and engineered to meet or exceed “five nines” 99.999% high availability standards, and provide very fast fault recovery through redundancy.

The rest of this paper is organized as follows. Section 2 describes background requirements of SandStone. Section 3 elaborates the detail design and section 4 presents the implementation. Section 5 details the experiments and insights gained by running SandStone in production. And we summarize the conclusion in Section 6.

2 Background

2.1 Scenario

In telecom network, there are a large number of subscriber profile data in a centralized way to store, such as IMPI/IMPU (IP Multimedia Private Identity/Public Identity) in IMS and CDR (Call Detail Record) in BSS (Billing Support System). Originally, SandStone was designed to implement basic IMS HSS (Home Subscriber Server) functional entity in a decentralized and self-organized way. As works going on, we found that SandStone holds much more potential for being used as a key-value storage system in most of the P2P telecom application scenarios referred in section 1. For the sake of clarity, we’ll choose the IMS HSS scenario as the application context.

According to 3GPP [11] specifications, HSS is the repository hosting the subscription related user data to support the other network entities such as CSCF (Call Session Control Function) to handle calls or sessions. Nowadays all the existing IMS HSS are centralized based. Typically it is hosted on expensive ATCA (Advanced Telecom Computing Architecture) or commercial super servers demanding a high robustness. The current HSS architecture works well on the deployment. Nevertheless, there are still some drawbacks other than the huge CAPEX (Capital Expenditure) and OPEX (Operating Expense). Firstly, central server is not a solution scaling well. Typically commercial HSSes are fully equipped nodes that can handle and store the data pertaining to a give maximum number of subscribers. Should the number of subscribers exceed the maximum limit, the operator is forced to deploy new HSS units that can handle the new subscriber data. Another question is the congested and overloaded servers can not be mitigated even there are spare resource somewhere else in the network. These drawbacks seriously hindered the development and deployment of IMS.

It is necessary to make some evolutions to adapt to the development trend in current network architectures. Although the distributed DHT based storage system seems quite capable to deal with these problems, the only doubt is whether it can achieve the same carrier grade performance goals as well as the central server based solutions.

2.2 Carrier Grade Objectives

According to current performance of HSS, SandStone for carrier grade of services has the following requirements:

Cost performance: Considering the saving of CAPEX and OPEX, SandStone is implemented on top of an infrastructure of tens of thousands of common computers located in many datacenters around the country. These common computers are very easy to maintain, add and replace than current ATCA servers.

Application Model: simple read and write operations to a data item that is uniquely identified by a key in SandStone. Every one million user corresponds to 1000 read and 100 write requests per second according to current application model of HSS. In other words, each peer in SandStone needs to support 10 read and 1 write requests per second at least in normal application model.

Scalability: the scalability in a self organized way, SandStone should be easily scaled up to handle 1 billion subscriber data. According to current HSS architecture, SandStone targets applications that need to store subscriber data that are relatively small. Each subscriber data should have a profile with 128K bytes, and tens of index mappings at the size of 128 bytes.

Reliability: Reliability is one of the most important requirements because even the slightest outage and congestion has significant financial consequences and impacts customers' experience. There are always a small but significant number of hosts that are failing at any given time. As such SandStone needs to be constructed in a manner that treats failure handling as the normal case without impacting availability or performance. The reliability objective is specified as "five nines" in telecom scopes.

Efficiency: For the best customers' quality of experience, any client operation should get the final response in 300 milliseconds, so as to limit the DHT lookup hops must be little with the presumption that SandStone will be deployed in a private network assuring 50 milliseconds is the maximum latency for any host-to-host pairs.

2.3 Failure Models

In a distributed system, the peer failure is a very common phenomenon. Peer's and network's failure models have an essential impact on the distributed storage systems, but this topic was rarely mentioned in above P2P inspired telecom research works.

We consider three failure models, including average failure (AF) and simultaneous failure (SBF1 and SBF2), respectively represents independent single computer fault, site/rack failure and inaccessible region due to IP backbone network break down.

Choosing exponential life distribution for a computer ($f(t)=\lambda e^{-\lambda t}$), we can get the failure rate as $\lambda(t) = 1/MTBF$ (Mean time between failure). From China present

mandatory standard, the personal common computer must have a MTBF no less than 4000 hours. We choose 1000 hours for the safe margin, this means, each computer has the probability of 0.001 to failure in one hour. Taking N_{pc} as the number of total computers in the DHT overlay network, then AF model will result in $N_{pc}/1000$ computers failure in every hour.

The site/rack failure, SBF1, is usually caused by power system blackouts or other unknown reasons in a site. It happens quite rarely in a telecom scene. We still choose exponential distribution to model it, with 20000 as the MTBF. In the actual deployment, it's up to 50 hosts in a site/rack, so SBF1 exhibits the scenario that 50 computers will get out of service simultaneously in each period of $20000*50/N_{pc}$ hours.

Finally, it is hard to estimate the probability distribution for SBF2. For example, maybe the backbones between regions are being destroyed by unexpected tornados or earthquakes. We transformed it into a design requirement; SandStone has to sustain one of the largest regions inaccessible. The “largest” means it may contain all computers of a certain region. After overcome these failure models above, this distributed storage system truly could be regarded as a carrier grade system.

3 The Design of SandStone

In this paper, we presented SandStone, a novel DHT key-value storage system with carrier grade performance which could be applied to telecom network architecture such as IMS. The main contributions of SandStone are as follows: a multi-layer DHT architecture that decreases the traffic overload in backbone network; a enhanced one-hop routing table that achieves rapid positioning of data resources; a data partition mechanism that provides N:N resources backup and parallel recovery; an adaptive write/read strategy and synchronization mechanism that guarantee high data consistency, and a high-performance disaster recovery strategy that maintains most of the resources available and reliable even when a certain region was disconnected. In the following, we describe the major design considerations of SandStone.

3.1 Architecture Overview

Fig.1 depicts the architecture of a SandStone peer. The SandStone is composed of four main components: data storage module, key based routing module, communication module and configurations & statistics module. Intercommunication between peers is completed by the bottom layer communication module.

Middle layer key based routing, KBR, refers to find the best suitable host for a key as the input, also include peers' ID allocation, DHT routing protocol, peers failure detecting and etc. In this layer, we implemented a novel ID allocation mechanism for traffic localization in section 3.2, and a one-hop DHT enhancement in section 3.3.

The top layer Data storage module takes charge of storage and management of subscriber data, include data storage, data restoring, data consistency verification and etc. Here we came up with a unique and practical replica placement strategy in section 3.4, and the enforced strong consistency strategy in section 3.5.

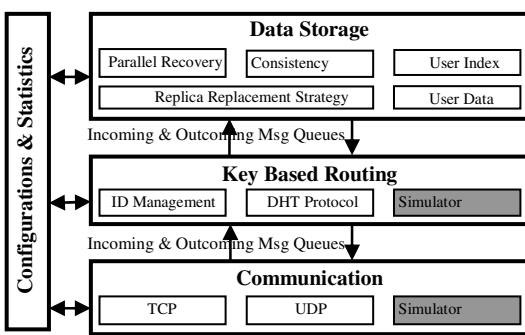


Fig. 1. The architecture of a SandStone peer

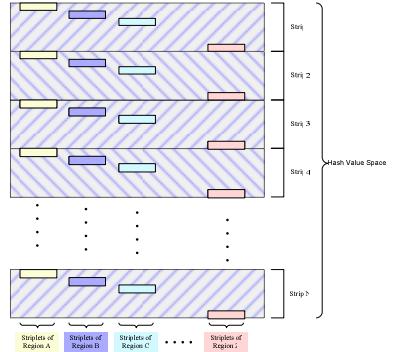


Fig. 2. The ID assignment

Furthermore, configurations & statistics module takes charge of managing and configuration the other layer modules. In addition, the simulation adaptor is introduced to let SandStone can be switched seamlessly between the simulation and realistic environment.

3.2 Traffic Localization

One of the key design requirements for SandStone is that it must make the traffic localization as far as possible. Recent studies [12, 13] have shown that a large volume of inter-domain redundant traffic by P2P mismatch problem already became the serious problem for ISPs. Due to the complexity of distributed networks, a novel traffic localization technique can not only consider the reduction of application response time but also decrease the backbone network overhead caused by inter-domain traffic.

Aiming at this goal, we have to answer two questions: (1) how to let peers carry region indication; (2) how to impel the data operations to exhibit a localized pattern.

First question, clearly, peer ID is an ideal place to embed regional identifiers such as province or city information. Most of solutions proposed to use a prefix for that, but such an ID assignment will result in severe load unbalance. SandStone use a Strip Segmentation solution as depicted in Fig.2. At first it divides the whole hash value space into N strips equally, and then it divides each strip into M striplets. M is the number of regions, and N can be user-defined, such as 1000. Notice that all the striplets at the same position of each strip constitutes the ID ranges for one region, peers belong to that region can randomly choose its ID within those intersected ranges. The Strip Segmentation can be regarded as a compromise to greatly mitigate the unbalance resulted by prefix solutions. This rule is known to every SandStone peer, thus it will be able to calculate the geographical region of any peer according to its ID.

For the second question, in the existing telecom network, subscriber data always carry the corresponding region information, such as subscriber attribution information, subscriber roaming information and etc. Utilized above characteristics, SandStone constructs a unique key from a subscriber data by using consistent hashing and with piggybacked its region information. Key k is assigned to the first peer whose identifier is equal to or follows k in the same region space. Every SandStone peer joins in the globe DHT and a logical region DHT at the same time, as shown in Fig.3.

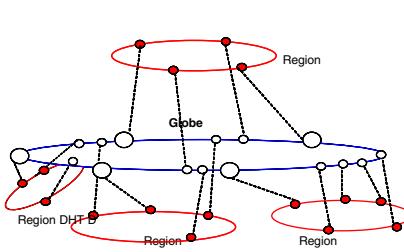


Fig. 3. The two layered DHT

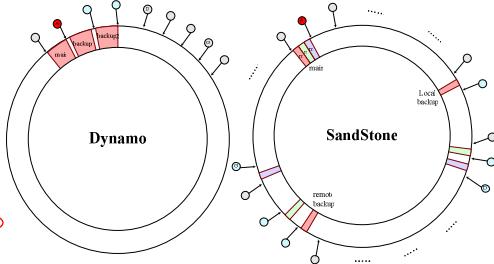


Fig. 4. The replica placement

Globe DHT and logical region DHTs are maintained by same KBR routing mechanisms. When a peer requests for a desired subscriber data, the default lookup strategy tries to search in the same region DHT as the requester's, then turn to the globe DHT if not found in region DHT. This simple design grasps some essence of the present user calling model, in which intra-region call attempts constitutes a major proportion, 70% for example. Because intra-region call/session handling related DHT operations will not bother the inter-region links anymore, it can be summarized that SandStone can achieve the same performance as traditional HSS in the aspect of traffic localization.

3.3 KBR Routing with One Hop Enhancement

In telecom networks, SandStone is built for latency sensitive applications that require at least 99.999% of read and write operations to be performed within 300 milliseconds. To meet these stringent latency requirements, it was imperative for us to avoid routing requests through multiple peers (which is the typical design adopted by several DHT systems such as Chord [14]). This is because multi-hop routing will bring on variability in response delay, increasing the latency at higher percentiles.

In order to meet the latency demand, SandStone can be characterized as a one-hop DHT, where each peer maintains enough routing information locally to route a request to the appropriate peer directly.

All SandStone peers belong to different regions form a globe ring topology just like Chord. In order to provide more efficient application response time, except the existing finger table router mechanism in Chord, SandStone makes a further step, to set up an extra One-hop Routing Table which include whole peers' router and state information for the one hop searching. One-hop Routing Table contains all the peers' router information in the DHT. Apparently the key question is that how to maintain the One-hop Routing Table freshness in every peer without too much resource consumption.

This is unrealistic that each peer maintains a One-hop Routing Table only by its self-organization mechanism on this large scale of network environment. This disorder organization and management method will cause a huge bandwidth overhead and routing table update latency. So there must be a number of special peers to play the role in unified managing and informing routing information. We will refer to these special peers which take charge of maintain routing information as SPM (Super Peer Maintenance).

There is at least one SPM node in certain region. All peers belong to this region must register to the local SPM when they joining into the SandStone system. The local SPM takes charge of peers in this region. SandStone failure detection mechanism uses a simple ping-style protocol that enables those peers in the system to learn about the arrival or departure of their neighbor peers. When joining or leaving the system, the peer's state change will be detected by its neighbor peer. And then the neighbor peer will inform local SPM this alternation information. A gossip-based protocol propagates peer alternation information and maintains a strongly consistent view of One-hop Routing Table between SPMs belongs to different regions. Finally, each SPM forward this notification in a broadcasting way to all peers which are located at its management region.

The advantage of SPM is that the routing information can be forwarded to all peers in entire system by a unified and effective fashion. It also decreases the traffic overhead in backbone network because one routing information message only transfers once in backbone links. It is worth mentioning that SPM nodes are only involved in routing table updating and managing in SandStone system. They don't deal with specific application requests so that they will not be the bottlenecks of the performance even with the increasing of application requests.

3.4 Replica Placement

To achieve high availability and durability, SandStone chooses the replica-based scheme that replicates the subscriber data on multiple peers. The replica placement policy serves two purposes: maximize data reliability and availability, and maximize network bandwidth utilization. For both, it is not enough to spread replicas across computers, which only guards against computer failures and fully utilizes each machine's network bandwidth. We must also spread data replicas across regions. This ensures that some replicas of a subscriber data will survive and remain available even if backbone network of a region is damaged in SBF2 scenario.

How to parallel recovery from the corresponding nodes is another important issue in replica mechanism of distributed storage system. BitVault and many related research works argued to spread the replicas randomly across the whole overlay, so as to make parallel recovery feasible. The drawback is that each peer must "remember" a long list of other peers hosting backups for its subscriber data, so make it not so scalability and reliability under churn. As shown in Fig.4, Dynamo still chose the original method to let the coordinator to replicate data at the $N-1$ clockwise successor nodes in the ring, N usually be 3. For a recovering operation, it is inefficient that node can only download data from the other corresponding $N-1$ nodes.

So in SandStone, we decided to choose three replicas as default R , represent the number of replicas for one subscriber data. Among the three replicas, the first would be stored in a peer of local region according to DHT key-value fundament, the second replica should be stored in a peer of different site but in same region, and the third would be maintained in the different region peer. This replicas' benefit is that not only improved traffic engineering based on specific application model (70% of applications are intra-region data operations), but also achieved good disaster recovery by out-regions.

At this time, new problems have emerged that how are replica locations selected? In SandStone, the backup data is stored in a certain segmentation rule so that it is no

need to remember the location of other replication data for any peers. To explain the replica placement, we need firstly define the ideal recovery factor L , as how many candidates should be used as the recovery source. Each peer calculates different offsets according to various data in its L triplet position. The keys of data are mapped to distinct strips after added different offsets, and then construct the backup keys of replica data. Finally, the replica data will be stored in different backup peers according to key-value principle and its backup key. Fig.4 depicts that a SandStone peer (red one) replicates its data partitions to 6 different peers, with $L=3$ and $R=3$. When red one recovering, it can pull the replication data from other 6 peers simultaneously. In this case, every peer only need to allocate one sixth space as the backup operation occupied for the failed peer, it's another benefit.

3.5 Consistency Strategy

To keep multiple replicas consistent is a difficult but inevitable task. Although data was partitioned, SandStone provides optimized eventual consistency, which allows for updates and amendments to be propagated to all replicas asynchronously.

Based on our specific application model, SandStone modifies the traditional “quorum” technique as its consistency strategy. For different peer failure modes, the read and write operation policies are defined as follows.

- (1) When there is no peer fail, according to our business mode that read operation is more than write operation (read is 10 times of write), SandStone implements $W=3$, $R=1$ strategy. In this case, the data must be successfully updated in 3 peers simultaneously; otherwise write operation returns fail and peers has been updated roll back the data. With respect to read operation, SandStone try to fetch main replication firstly, if it is ok, then return the data to application client, otherwise it try to fetch local backup replication and remote backup replication in sequence. Only all three operations failed, this read operation return fails;
- (2) When there are some peers fails, considering there is a trade-off between availability and consistency, SandStone implements an adaptive write policy for different data operation type. The “delete” operation still enforces of $W=3$ strategy; the “add” operation enforces of $W \geq 2$ strategy, and the “modify” operation enforces of $W \geq 1$ strategy. So each operation has a minimal acceptable number X (for delete/add/modify operation, the X is 3/2/1 respectively). Assuming, for one given data, Y is the number of current available peers which are responsible for storing three copies of the data. If $Y \geq X$, then $W=Y$ strategy is enforced; otherwise this data operation return fail. With respect to read operation, $R=1$ strategy is still enforced.
- (3) When a fail peer comes back and under restoring process which synchronizes data from corresponding peers, the read operation will be blocked by this peer and return fail.
- (4) If a peer joins, the data which the new peer is responsible for storing need to be moved from successor node. After that the incremental data restore process will be triggered and the corresponding data will be transmitted to the new arrived peer. On the other hand, the read operation on this peer will be blocked until the restoring process finished. If a peer leaves, the successor node will perform incremental data restore process for the new incremental key range space. Then the read operation within the incremental key range on this peer will be blocked until the restoring process finished.

In the aspect of data version, with the introduction of NTP servers, SandStone uses timestamp as version tag in order to capture causality between different versions of the same data. One can determine which version is the latest one by examine their timestamps. If the version of same data in corresponding peer conflict, then the older one will be overwritten and updated.

Base on consistency strategy described above, SandStone provides eventual consistency guarantee which has been proved in a wide range of experiments.

4 Implementation

We implemented SandStone in C++, atop the ACE [15] (Adaptive Communication Environment) toolkit which provides a rich set of reusable C++ wrapper facades and framework components that perform common communication software tasks across a range of OS platforms. In the choice of the storage engines for SandStone, we have two kinds of options. One is an open source database MySQL [16] which is easy to development and integration. On the other hand, we also implemented a simplified memory database MDB to improve the efficiency of data access.

We replaced the existing HSS module by our SandStone in IMS system. The architecture of IMS over SandStone is shown in Fig.5. A-CSCF is combined by P-CSCF, I-CSCF and S-CSCF for the flat structure design. Toward A-CSCF module, it doesn't need to know either HSS or SandStone is bottom subscriber database. The A-CSCF module access SandStone peers through existing IMS Cx interface and lookups the subscriber data in whole DHT network. When SandStone peers found the target data successfully, it returns the data to A-CSCF module.

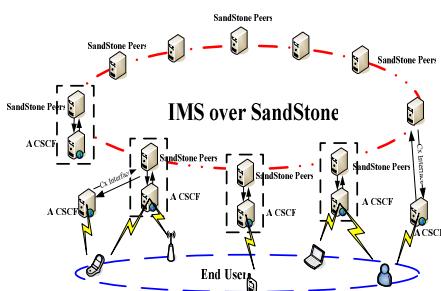


Fig. 5. The architecture of IMS over SandStone

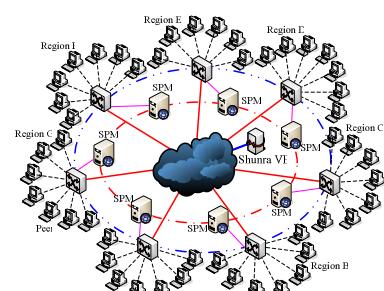


Fig. 6. The topology of SandStone

Furthermore, in order to simulate the various scenarios and the business modes for evaluating the performance of SandStone, we implemented two auxiliary testing tools TT (Testing Tool) and PTT (Performance Testing Tool) in addition. TT is a comprehensive test and management tool which can control the behavior of peers, trigger various peers failure model, and monitor the state of whole peers such as their routing table information. PTT is another test tool for performance evaluation. For simulating the CSCF module, it can send various application requirement packets by introducing different business scene model.

5 Experiments

In the process of building and deploying SandStone, we have experienced a variety of issues, some operational and technical. We built a SandStone prototype of 3000 peers located in three regions separately, each of which is a commodity PC with a 2.2GHz Core2 CPU and 2GB memory. The operating system on each peer is SUSE9 sp4, running the Linux 2.6.5-7.308 kernel. We use two experimental network environments to measure our system. The first is that these computers are connected with a series of Huawei Quidway S3500 1000Mb Ethernet switches. The second one introduces a Shunra Virtual Enterprise [17] to simulate a wide variety of network impairments, exactly as if SandStone were running in a real production environment. The topology is shown in Fig.6. Unless otherwise specified, the experiments lasted 24 hours.

5.1 Load Balance

In order to determine whether a distributed storage system stores data rationally and effectively, we first measure data load distribution in SandStone. We randomly generated 10 million subscriber data according to user registered statistics by China Mobile, and put them into the 3000 peers based on DHT key-value rule. Finally, we made a statistic for distribution of these data in 3000 peers.

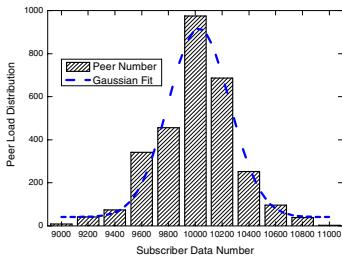


Fig. 7. Load balance

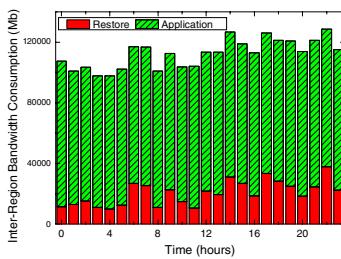


Fig. 8. The inter-region bandwidth consumption

From the Fig.7, through the optimization of strip segmentation arithmetic and ID assignment mechanism, the subscriber data obeys a Gaussian-like distribution. The number of data in 98% of peers maintained at around 10000. Data stored in SandStone have perfect load-sharing so that it also improves the traffic balance finally.

5.2 Reliability

To achieve high availability and durability, SandStone uses strict replica placement and consistency strategy. In this scenario, we evaluate the reliability of SandStone in the face of a variety of failure and application models.

As shown in table 1, under normal application model, each peer received 1 write and 10 read requests per second. 1% AF model will result in 1% of peers fail in every

hour, and in the next hour 80% of fail peers will be comeback. As is clear from table 1, the 1% and 3% AF model almost no impact on business success rate. The success rate in 12 hours is almost 100%. The SBF1 and SBF2 model lead to more number of peers' failure, but the SandStone still able to maintain more than 99.9% success rate. With the increased of business requests, the success rate has declined, but still more than 99%.

Table 1. The reliability of SandStone

Application Model	Failure Model	Success Rate
write:1/s per peer	AF (1%)	≈100%
	AF (3%)	≈100%
read:10/s per peer	SBF1	99.9992%
	SBF2	99.951%
write:30/s per peer	AF (1%)	99.99995%
	AF (3%)	99.9997%
read:50/s per peer	SBF1	99.9959%
	SBF2	99.68%
write:50/s per peer	AF (1%)	99.99992%
	AF (3%)	99.9973%
read:100/s per	SBF1	99.9934%

Table 2. The recovery time in SandStone

Database Mode	Peer Behavior	Recovery Time (seconds)
MySQL	Inactive (30 mins)	40.1
	Inactive (60 mins)	75.2
	Add	161.2
	Delete	82.6
Memory DB	Inactive (30 mins)	31.7
	Inactive (60 mins)	58.1
	Add	132.7
	Delete	74.1

5.3 Latency

While SandStone's principle design goal is to build a highly available data storage system, application response time is an equally important criterion. As mention above, to provide a best customer experience, SandStone must guarantees that the application can deliver its functionality in a bounded time. In this experiment, we measured the application response latency under various application models' frequency.

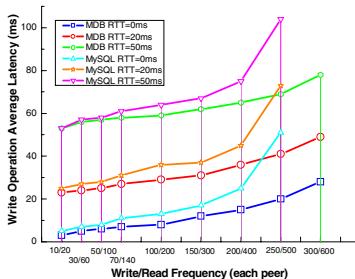


Fig. 9. Write operation average latency

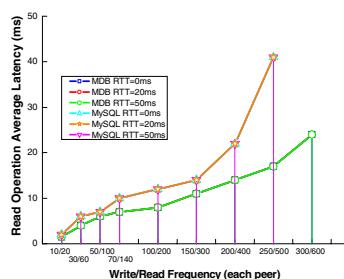


Fig. 10. Read operation average latency

It can be seen from the Fig.9 and Fig.10, with the expanding of business requests, the latency of write and read requests are increased. But even in the most unexpected application model situation (write/read frequency is 300/600 per second per peer); the latency is also much lower than the restriction of 300ms which is the business can tolerate. In the follow-up experiment, we turn on the Shunra Virtual Enterprise to

simulate the path propagation delay between different regions, from 0ms to 50ms. At this time we can see from the figure, with the increased propagation delay, the latency of write request operation has a greater degree of change obviously because of remote data replica placement strategy in section 3.4. However, because most of the data can be obtained from the local peers due to data localization, the latency of read request operations is basically no change, although frequency is increased. Furthermore, it is evident that the efficiency of Memory DB is higher than MySQL and the latency under Memory DB environment is lower than under MySQL's, especially in heavy business requests models. It is worth mentioning that the system can't afford the 300/600 frequency under MySQL environment because of the limitation of DB pools.

5.4 Recovery Time

In SandStone system, peer outages (due to failures and maintenance tasks) are often transient but may last for extended intervals. For systems prone to peer and network failures, availability can be increased by using optimistic replication techniques, where data are allowed to propagate to replicas in the background, and concurrent, disconnected work is tolerated. When peer come back again, it will recover incremental data from other replicas. So data recovery process time also determines the reliability of the system. In this experiment, we compared the data recovery time under series of node-state changes scenario. In experiment environment, each peer have stored approximately 10000 subscriber data, and the application model is normal, so each peer received 1 write and 10 read requests per second.

From the table 2, as a result of the parallel synchronization and incremental recovery technology, peer data recovery time maintained at a very small value. Even in the Add and Delete scenario that need to synchronize the entire data, a peer change process only consumed less than 200 second. This kind of recovery time in the Carrier Grade network is totally acceptable. Meanwhile, it is clear that the recovery time of Memory DB is shorter than MySQL's.

5.5 Bandwidth Consumption

Although SPM node is only responsible for the maintenance and synchronization of one-hop routing table, its bandwidth consumption is still one of our concerned issues. We recorded the bandwidth consumption of SPM under various failure models.

As can be seen in table 3, in the general failure model such as AF model, the bandwidth consumption of SPM is very tiny, almost can be ignored. Even in the case of SBF1 model (almost 50% of peers in same region failed simultaneously); the bandwidth consumption is just 820KB/s. For such a very low probability of unexpected events, this overhead of SPM is totally acceptable. It is worth mentioning that SPM uses a unified manner to inform the routing table change in SBF2 model, so the bandwidth consumption maintained at a very small value.

On the other hand, we calculate the inter-region bandwidth consumption in whole SandStone network. The application model is normal, and 3% AF failure model is introduced. From the Fig.8, we can see that despite the restore traffic in the ever-changing, but the overall inter-region bandwidth consumption is still maintained at a relatively stable value that decreases the impact to backbone network.

Table 3. The bandwidth consumption of SPM

Failure Model	Bandwidth Consumption (KB/S)
AF (1%)	1.9
AF (5%)	34.8
AF (10%)	75.6
SBF1 (50fails)	93
SBF1 (100fails)	210
SBF1 (200fails)	450
SBF1 (500fails)	820
SBF2	15.3

6 Conclusion

In this paper, we present the analysis and main design considerations of SandStone. The main contributions of SandStone are: a one-hop DHT enhancement, a Strip Segmentation ID assignment and a two layered DHT for traffic localization, a novel replica placement schemes and an enhanced protocol for data consistency strategy. Till now, by simulation and running in an experimental environment (thousands of nodes), SandStone achieved the Carrier Grade performance as listed in section 2.2. We'll seek more deployments to testify the SandStone performance, and adapt SandStone to more telecom application scenarios as listed in section 1.

References

1. P2PSIP (2009), <http://www.ietf.org/html.charters/p2psip-charter.html>
2. Matuszewskil, M., Garcia-Martin, M.A.: A Distributed IP Multimedia Subsystem (IMS). In: Proc. of WoWMoM (2007)
3. Van Leekwijck, W., Tsang, I.-J.: Distributed Multimedia Control System Using an In-network Peer-to-Peer Architecture. In: Proc. of ICIN (2006)
4. Ghemawat, S., Gobioff, H., Leung: The Google file system. In: Proc. of ACM SOSP (2003)
5. Oceanstore (2009), <http://oceanstore.cs.berkeley.edu>
6. Rhea, S., Eaton, P., Geels, D., Weatherspoon, H., Zhao, B., Kubiatowicz, J.: Pond: the oceanstore prototype. In: Proc. of USENIX FAST (2003)
7. Zhang, Z., Lian, Q., Lin, S., Chen, W., Chen, Y., Jin, C.: BitVault: a highly reliable distributed data retention platform. ACM SIGOPS Operating Systems Review archive 41(2), 27–36 (2007)
8. Weil, S.A., Brandt, S.A., Miller, E.L., Long, D.D.E., Maltzahn, C.: Ceph, A scalable, high-performance distributed file system. In: Proc. of OSDI (2006)
9. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G.: Dynamo: Amazon's Highly Available Key-value Store. In: Proc. of ACM SOSP (2007)
10. Stribling, J., Sovran, Y., Zhang, I., Pretzer, X., Li, J., Kaashoek, F., Morris, R.: Simplifying Wide-Area Application Development with WheelFS. In: Proc. of USENIX NSDI (2009)
11. 3GPP (2009), <http://www.3gpp.org/>

12. Aggarwal, V., Feldmann, A., Scheideler, C.: Can ISPs and P2P Systems Cooperate for Improved Performance? *ACM SIGCOMM Computer Communications Review* 37(3), 29–40 (2007)
13. Shen, G., Wang, Y., Xiong, Y., Zhao, B., Zhang, Z.: HPTP: Relieving the Tension between ISPs and P2P. In: Proc. of USENIX IPTPS (2007)
14. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In: Proc of ACM SIGCOMM 2001, San Deigo, CA, August 2001, pp. 149–160 (2001)
15. ACE (2009), <http://www.cs.wustl.edu/~schmidt/ACE.html>
16. MySQL (2009), <http://www.mysql.com/>
17. Shunra, V.E.: (2009), <http://www.shunra.com/>

Selective Replicated Declustering for Arbitrary Queries

K. Yasin Oktay, Ata Turk, and Cevdet Aykanat

Bilkent University, Department of Computer Engineering,

06800, Ankara, Turkey

`koktay@ug.bilkent.edu.tr, {atat,aykanat}@cs.bilkent.edu.tr`

Abstract. Data declustering is used to minimize query response times in data intensive applications. In this technique, query retrieval process is parallelized by distributing the data among several disks and it is useful in applications such as geographic information systems that access huge amounts of data. Declustering with replication is an extension of declustering with possible data replicas in the system. Many replicated declustering schemes have been proposed. Most of these schemes generate two or more copies of all data items. However, some applications have very large data sizes and even having two copies of all data items may not be feasible. In such systems selective replication is a necessity. Furthermore, existing replication schemes are not designed to utilize query distribution information if such information is available. In this study we propose a replicated declustering scheme that decides both on the data items to be replicated and the assignment of all data items to disks when there is limited replication capacity. We make use of available query information in order to decide replication and partitioning of the data and try to optimize aggregate parallel response time. We propose and implement a Fiduccia-Mattheyses-like iterative improvement algorithm to obtain a two-way replicated declustering and use this algorithm in a recursive framework to generate a multi-way replicated declustering. Experiments conducted with arbitrary queries on real datasets show that, especially for low replication constraints, the proposed scheme yields better performance results compared to existing replicated declustering schemes.

1 Introduction

Data declustering is one of the key techniques used in management of applications with humongous-scale data processing requirements. In this technique, query retrieval process is parallelized by distributing the data among several disks. The most crucial part of exploiting I/O parallelism is to develop distribution techniques that enable parallel access of the data. The distribution has to respect disk capacity constraints while trying to locate data items that are more likely to be retrieved together into separate disks.

There are many declustering schemes proposed for I/O optimization (See [\[1\]](#) and the citations contained within), especially for range queries. These schemes generally try to scatter neighboring data items into separate disks.

There are also a few studies that propose to exploit query distribution information [2], [3], [4], if such information is available. For equal-sized data items, the total response time for a given query set can be minimized by evenly distributing the data items requested by each query across the disks as much as possible, while taking query frequencies into consideration. In [3], the declustering problem with a given query distribution is modeled as a max-cut partitioning of a weighted similarity graph. Here, data items are represented as vertices and an edge between two vertices indicate that corresponding data items appear in at least one query. The edge weights represent the likelihood that the two data items represented by the vertices of the edge will be requested together by queries. Hence, maximizing the edge cut in a partitioning of this similarity graph relates to maximizing the chance of assigning data items that will probably appear in the same query to separate disks. In [2] and [4], the deficiencies of the weighted similarity graph model are addressed and a correct hypergraph model which encodes the total I/O cost correctly is proposed. In this representation, vertices represent data items and hyperedges/nets represent queries, where each net representing a query connects the subset of vertices that corresponds to the data items requested by that query. The vertex weights represent the data item sizes and net weights represent the query frequencies. Recently, hypergraph models have also been applied for clustering purposes in data mining ([5]) and road network systems ([6], [7]).

In addition to declustering, replication of data items to achieve higher I/O parallelism has started to gain attention. There are many replicated declustering schemes proposed for optimizing range queries (See [8] and the citations contained within). Recently, there are a few studies that address this problem for arbitrary queries as well [9], [10], [11]. In [9], a Random Duplicate Assignment (RDA) scheme is proposed. RDA stores a data item on two disks chosen randomly from the set of disks and it is shown that the retrieval cost of random allocation is at most one more than the optimal with high probability. In [10], Orthogonal Assignment (OA) is proposed. OA is a two-copy replication scheme for arbitrary queries and if the two disks that a data item is stored at are considered as a pair, each pair appears only once in the disk allocation of OA. In [11], Design Theoretic Assignment (DTA) is proposed. DTA uses the blocks of an $(K, c, 1)$ design for c -copy replicated declustering using K disks. A block and its rotations can be used to determine the disks on which the data items are stored.

Unfortunately, none of the above replication schemes can utilize query distribution information if such information is available. However, with the increasing and extensive usage in GIS and spatial database systems, such information is becoming more and more available, and it is desirable for a replication scheme to be able to utilize this kind of information. Furthermore, replication has its own difficulties, mainly in the form of consistency considerations, that arise in update and deletion operations. Response times for write operations slow down when there is replication. Finally, with replication comes higher storage costs and there are applications with very large data sizes where even two-copy replication

is not feasible. Thus, if possible, unnecessary replication has to be avoided and techniques that enable replication under given size constraints must be studied.

In this study, we present a selective replicated declustering scheme that makes use of available query information and optimizes aggregate parallel response time within a given constraint on the replication amount due to disk size limitations. In the proposed scheme, there is no restriction on the replication counts of individual data items. That is, some data items may be replicated more than once while some other data items may not even be replicated at all. We propose an iterative-improvement-based replication algorithm that uses similar data structures with the Fiduccia-Mattheyses (FM) Heuristic [12] and recursively bipartition and replicate the data. FM is a linear time iterative improvement heuristic which was initially proposed and used for clustering purposes in bipartitioning hypergraphs that represent VLSI circuits. The neighborhood definition is based on single-vertex moves considered from one part to the other part of a partition. FM starts from a random feasible bipartition and updates the bipartition by a sequence of moves, which are organized as passes over the vertices. In [2], the authors propose an FM-like declustering heuristic without replication.

The rest of the paper is organized as follows. In Section 2, necessary notations and formal definition of the replication problem is given. The proposed scheme is presented in Section 3. In Section 4, we experiment and compare our proposed approach with two replications schemes that are known to perform good on arbitrary queries.

2 Notation and Definitions

Basic notations, concepts and definitions used throughout the paper are presented in this section. We are given a dataset D with $|D|$ indivisible data items and a query set Q with $|Q|$ queries, where a query $q \in Q$ requests a subset of data items, i.e., $q \subseteq D$. Each query q is associated with a relative frequency $f(q)$, where $f(q)$ is the probability that query q will be requested. We assume that query frequencies are extracted from the query log and future queries will be similar to the ones in the query log. We also assume that all data items and all disks are homogeneous and thus, the storage requirement and the retrieval time of all data items on all disks are equal and can be accepted as one for practical purposes.

Definition 1. *Replicated Declustering Problem: Given a set D of data items, a set Q of queries, K homogeneous disks with storage capacity C_{max} , and a maximum allowable replication amount c , find K subsets of D (or a K -way replicated declustering of D), say $R_K = \{D_1, D_2, \dots, D_K\}$, which, if assigned to separate disks, minimizes the aggregate response time $T(Q)$ for Q and satisfies the following feasibility conditions:*

- i. $\bigcup_{k=1}^K D_k = D$
- ii. $\sum_{k=1}^K |D_k| \leq (1 + c) \times |D|$ and
- iii. $|D_k| \leq C_{max}$.

Given a multi-disk system with replicated data, the problem of finding an optimal schedule for retrieving the data items of a query arises. The optimal schedule for a query minimizes the maximum number of data items requested from a disk. This problem can be solved optimally by a network-flow based algorithm [13]. Hence, given a replicated declustering R_K and a query q , optimal scheduling $S(q)$ for q can be calculated. $S(q)$ indicates which copies of the data items will be accessed during processing q . So $S(q)$ can be considered as partitioning q into K disjoint subqueries $S(q) = \{q_1, q_2 \dots q_K\}$ where q_k indicates the data items requested by q from disk D_k .

Definition 2. *Given a replicated declustering R_K , a query q and an optimal schedule $S(q) = \{q_1, q_2 \dots q_K\}$ for q , response time $r(q)$ for q is: $r(q) = \max_{1 \leq k \leq K} \{t_k(q)\}$, where $t_k(q) = |q_k|$ denotes the total retrieval time of data items on disk D_k that are requested by q .*

Definition 3. *In a replicated declustering R_K , the aggregate parallel response time for a query set Q is $T(Q) = \sum_{q \in Q} f(q)r(q)$.*

3 Proposed Approach

Here, we first describe a two-way replicated declustering algorithm and then show how recursion can be applied on top of this two-way replicated declustering to obtain a multi-way replicated declustering. Our algorithm starts with a randomly generated initial feasible two-way declustering of D into D_A and D_B , and iteratively improves this two-way declustering by move and replication operations. Since there are replications, there are three states that a data item can be in: A , B , and AB , where A means that the data item is only in part D_A , B means that the data item is only in part D_B , and AB means that the data item is replicated. Furthermore, for the data items requested by each query, we keep track of the number of data items in each part. That is, $t_A(q)$ indicates the number of data items requested by query q that exists only in part D_A , $t_B(q)$ indicates the number of data items requested by query q that exists only in part D_B , and $t_{AB}(q)$ indicates the number of data items requested by query q that are replicated. Note that the total number of data items requested by query q is equal to $|q| = t_A(q) + t_B(q) + t_{AB}(q)$.

In Algorithm 1, we initialize move gains and replication gains for each data item. First, for each query q , we count the number of data items that are in A , B and AB state among the data items requested by q (lines 1–5). Here, $State$ is a vector which holds the states of the data items, i.e., $State(d)$ stores the current state of data item d . Then, we calculate the move gain $g_m(d)$ and the replication gain $g_r(d)$ of each data item (6–17). Note that only non-replicated data items are amenable to move and replication. So, for a non-replicated data item d , $State(d)$ denotes the source part for a possible move or replication associated with data item d . For a non-replicated data item d in state A , the associated move operation changes its state to B , whereas the associated replication operation changes its state to AB . A dual discussion holds for a data item d that is in state B . The for loop in lines 10–18

Algorithm 1. InitializeGains($(\mathcal{D}, \mathcal{Q})$, $\Pi_2 = \{\mathcal{D}_A, \mathcal{D}_B\}$)

Require: $(\mathcal{D}, \mathcal{Q})$, $\Pi_2 = \{\mathcal{D}_A, \mathcal{D}_B\}$

- 1: **for each** query $q \in \mathcal{Q}$ **do**
- 2: $t_A(q) \leftarrow t_B(q) \leftarrow t_{AB}(q) \leftarrow 0$
- 3: **for each** data item $d \in q$ **do**
- 4: $s \leftarrow \text{State}(d)$
- 5: $t_k(q) \leftarrow t_k(q) + 1$
- 6: **for each** non-replicated data item $d \in \mathcal{D}$ **do**
- 7: $g_m(d) \leftarrow g_r(d) \leftarrow 0$
- 8: $s \leftarrow \text{State}(d)$
- 9: **for each** query q that contains d **do**
- 10: $\Delta \leftarrow \text{DeltaCalculation}(q, \text{State}(d))$
- 11: **if** $\Delta \geq 2$ **then**
- 12: $g_m(d) \leftarrow g_m(d) + f(q)$
- 13: $g_r(d) \leftarrow g_r(d) + f(q)$
- 14: **else if** $(\Delta = 0) \wedge (2(t_k(q) + t_{AB}(q)) = |q|)$ **then**
- 15: $g_m(d) \leftarrow g_m(d) - f(q)$
- 16: **else if** $\Delta \leq -1$ **then**
- 17: $g_m(d) \leftarrow g_m(d) - f(q)$

computes and uses a Δ value for each query that requests d , where Δ represents the difference between the number of data items of q in the source and destination parts under an optimal schedule of query q across these two parts. Algorithm 2 shows the pseudocode for Δ calculation. In lines 1–5 of Algorithm 2, we calculate Δ when some of the replications are unnecessary for query q . This means that some of the replicated data items will be retrieved from the source part s , while others will be retrieved from the other part for q . In this case, if the required number of data items for that query is odd, Δ will be 1, and it will be 0 otherwise. Lines 6–10 indicate that all replications will be retrieved from only one part. We first check the case that all replications are retrieved from the given part s (lines 7–8) and in lines 9–10 we handle the other case.

Algorithm 2. DeltaCalculation(q, s).

Require: $(q \in \mathcal{Q}, s)$

- 1: **if** $|t_A(q) - t_B(q)| < t_{AB}(q)$ **then**
- 2: **if** $|q| \% 2 = 0$ **then**
- 3: $\Delta \leftarrow 0$
- 4: **else**
- 5: $\Delta \leftarrow 1$
- 6: **else**
- 7: **if** $t_s(q) < |q| - t_{AB}(q) - t_s(q)$ **then**
- 8: $\Delta \leftarrow 2 \times t_s(q) + 2 \times t_{AB}(q) - |q|$
- 9: **else**
- 10: $\Delta \leftarrow 2 \times t_s(q) - |q|$

Algorithm 3. Update gains after a move from A to B .

Require: $(\mathcal{D}, \mathcal{Q})$, $\Pi_2 = \{\mathcal{D}_A, \mathcal{D}_B\}$, $d^* \in \mathcal{D}_A$

- 1: **for each** query $q \in \mathcal{Q}$ that contains d^* **do**
- 2: $\Delta \leftarrow \text{DeltaCalculation}(q, A)$
- 3: **for each** non-replicated data item $d \in q$ **do**
- 4: **if** $d \in \mathcal{D}_A$ **then**
- 5: **if** $\Delta = 3$ **then**
- 6: $g_m(d) \leftarrow g_m(d) - f(q)$
- 7: $g_r(d) \leftarrow g_r(d) - f(q)$
- 8: **else if** $\Delta = 2$ **then**
- 9: $g_r(d) \leftarrow g_r(d) - f(q)$
- 10: **if** $t_{AB}(q) \geq 1$ **then**
- 11: $g_m(d) \leftarrow g_m(d) - f(q)$
- 12: **else**
- 13: $g_m(d) \leftarrow g_m(d) - 2f(q)$
- 14: **else if** $\Delta = 1 \wedge t_A + t_{AB}(q) = t_B + 1$ **then**
- 15: $g_m(d) \leftarrow g_m(d) - f(q)$
- 16: **else if** $\Delta = 0 \wedge |q| = 2(t_B(q) + 1)$ **then**
- 17: $g_m(d) \leftarrow g_m(d) - f(q)$
- 18: **else if** $d \in \mathcal{D}_B$ **then**
- 19: **if** $\Delta = 1 \wedge (t_A(q) - t_B(q) = t_{AB}(q) + 1)$ **then**
- 20: $g_m(d) \leftarrow g_m(d) + f(q)$
- 21: **else if** $\Delta = 0$ **then**
- 22: **if** $t_{AB}(q) = 0$ **then**
- 23: $g_m(d) \leftarrow g_m(d) + 2f(q)$
- 24: $g_r(d) \leftarrow g_r(d) + f(q)$
- 25: **else if** $|t_A(q) - t_B(q)| = t_{AB}(q)$ **then**
- 26: $g_m(d) \leftarrow g_m(d) + f(q)$
- 27: **if** $t_B(q) - t_A(q) = t_{AB}(q)$ **then**
- 28: $g_r(d) \leftarrow g_r(d) + f(q)$
- 29: **else if** $\Delta = -1$ **then**
- 30: $g_m(d) \leftarrow g_m(d) + f(q)$
- 31: $g_r(d) \leftarrow g_r(d) + f(q)$
- 32: $t_A(q) \leftarrow t_A(q) - 1$
- 33: $t_B(q) \leftarrow t_B(q) + 1$
- 34: $\text{State}(d^*) \leftarrow B$
- 35: $\text{Locked}(d^*) \leftarrow 1$

In Algorithms 3 and 4, we update the move and replication gains of the unlocked data items after the tentative move or replication of data item d^* from the source part A to the destination part B , respectively. The dual of these algorithms which performs moves or replications from B to A are easy to deduce from Algorithms 3 and 4. We just update the gains of the data items that share at least one query with the moved or replicated data item d^* . Selection of the operation with maximum gain necessitates maintaining two priority queues, one for moves, one for replications, implemented as binary max-heaps in this work. The priority queue should support extract-max, delete, increase-key and decrease-key operations.

The overall algorithm can be summarized as follows. The algorithm starts from a randomly constructed initial feasible two-way declustering. The

Algorithm 4. Update gains after a replication from A to B .

Require: $(\mathcal{D}, \mathcal{Q})$, $\Pi_2 = \{\mathcal{D}_A, \mathcal{D}_B\}$, $d^* \in \mathcal{D}_A$

- 1: **for each** query $q \in \mathcal{Q}$ that contains d^* **do**
- 2: $\Delta \leftarrow \text{DeltaCalculation}(q, A)$
- 3: **for each** non-replicated data item $d \in q$ **do**
- 4: **if** $d \in \mathcal{D}_A$ **then**
- 5: **if** $\Delta = 3 \vee \Delta = 2$ **then**
- 6: $g_m(d) \leftarrow g_m(d) - f(q)$
- 7: $g_r(d) \leftarrow g_r(d) - f(q)$
- 8: **else if** $d \in \mathcal{D}_B$ **then**
- 9: **if** $\Delta = 1 \wedge (t_A(q) - t_B(q) = t_{AB}(q) + 1)$ **then**
- 10: $g_m(d) \leftarrow g_m(d) + f(q)$
- 11: **else if** $\Delta = 0 \wedge (t_A(q) - t_B(q) = t_{AB}(q))$ **then**
- 12: $g_m(d) \leftarrow g_m(d) + f(q)$
- 13: $t_A(q) \leftarrow t_A(q) - 1$
- 14: $t_{AB}(q) \leftarrow t_{AB}(q) + 1$
- 15: $\text{State}(d^*) \leftarrow AB$
- 16: $\text{Locked}(d^*) \leftarrow 1$

initial move and replication gains are computed using the algorithm shown in Algorithm 1. At the beginning of each pass, all data items are unlocked. At each step in a pass, an unlocked data item with maximum move or replication gain (even if it is negative), which does not violate the feasibility conditions, is selected to be moved or replicated to the other part and then it is locked and removed from the appropriate heaps. If maximum move and replication gains are equal, move operation is preferred. If the maximum gain providing operation is an infeasible replication, we trace all replicated data items to see if there are unnecessary replications. If this is the case, we delete those data items to see whether the subject replication operation becomes feasible. We adopt the conventional locking mechanism, which enforces each data item to be moved or replicated at most once during a pass, to avoid thrashing. After the decision of the move or replication operation, the move and replication gains of the affected data items are updated using Algorithms 3 and 4. The change in total cost is recorded along with the performed operation. The pass terminates when no feasible operation remains. Then, the initial state before the pass is recovered and a prefix subsequence of operations, which incurs the maximum decrease in the cost, is performed.

We applied the proposed replicated two-way declustering algorithm in a recursive framework to obtain a multi-way replicated declustering. In this framework, every two-way replicated declustering step for a database system (D, Q) generates two database sub-systems (D_A, Q_A) and (D_B, Q_B) . We should note here that, since we can perform replication, $|D_A| + |D_B| \geq |D|$. Since we delete all unnecessary replications at the end of each pass, all replicated data items are necessary in both parts. Thus, all data items in state A and AB are added into D_A , whereas all data items in state B and AB are added into D_B . In order to

perform recursive replicated declustering, splitting of queries is a necessity as well. Each query q is split into two sub-queries depending on the two-way declustering and the optimal schedule for that query. So, the objective at each recursive two-way declustering step models the objective of even distribution of queries into K disks. We only discuss recursive declustering for the case when K is a power of two. However, the proposed scheme can be extended for arbitrary K values by enforcing properly imbalanced two-way declusterings. For $K = 2^\ell$, the storage capacity at the i th recursion level is set to be $C_{max} \times (K/2^i)$ for $i = 1, \dots, \ell$. In our current implementation, the global maximum allowable replication amount c is applied at each recursive step where each replication operation reduces it by one and each deletion increases it by one.

4 Experimental Results

In our experiments, we used three of the datasets used in [2] along with the synthetically created arbitrary query sets. We used homogeneous data sizes, equal query frequencies, and homogeneous servers. We tested the proposed Selective Replicated Declustering (SRD) algorithm on these datasets and compared SRD with the RDA and OA algorithms. These algorithms are known to perform good for arbitrary queries and they can support partial replication. All algorithms are implemented in C language on a Linux platform.

Table 1 shows the properties of the three database systems used in the experiments. Further details about these datasets can be found in [2]. We have tested all of our datasets under varying replication constraints.

Figs. 11 and 12 display the variation in the relative performances of the three replicated declustering algorithms with increasing replication amounts for $K = 16$ and $K = 32$ disks, respectively. For the RDA and OA schemes, the data items to be replicated are selected randomly. In Figs. 11 and 12, the ideal response time refers to the average parallel response time of a strictly optimal declustering if it exists. So, it is effectively a lower bound for the optimal response time. Note that a declustering is said to be strictly optimal with respect to a query set if it is optimal for every query in the query set. A declustering is optimal for a query $q \in Q$, if the response time $r(q)$ is equal to $\lceil |q|/K \rceil$, where K is the number of disks in the system.

The relative performance difference between the replicated declustering algorithms decreases with increasing amount of replication as expected. The

Table 1. Properties of database systems used in experiments (taken from [2])

Dataset	$ \mathcal{D} $	$ \mathcal{Q} $	Average query size
HH	1638	1000	43.3
FR	3338	5000	10.0
Park	1022	2000	20.1

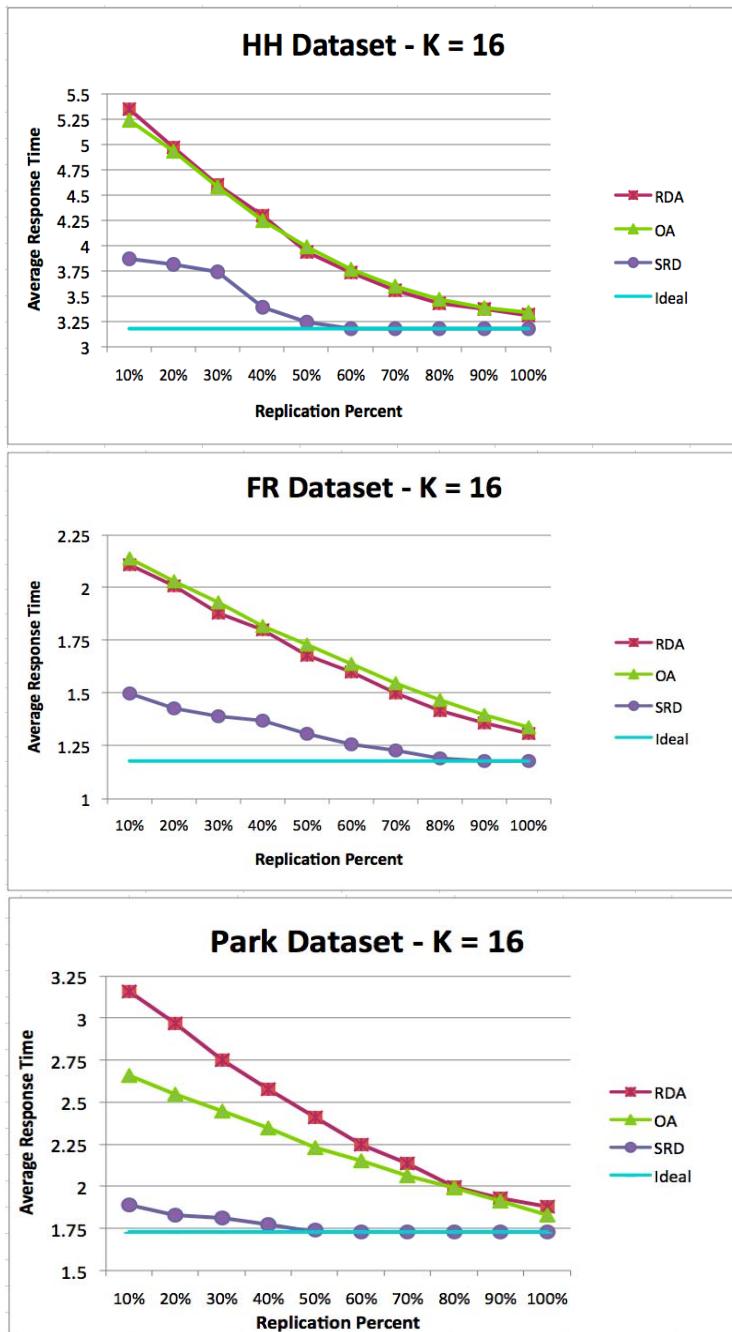


Fig. 1. Comparison of average response time qualities of the Random Duplicate Assignment (RDA), Orthogonal Assignment (OA), and Selective Replicated Declustering (SRD) schemes with increasing replication for $K = 16$ disks

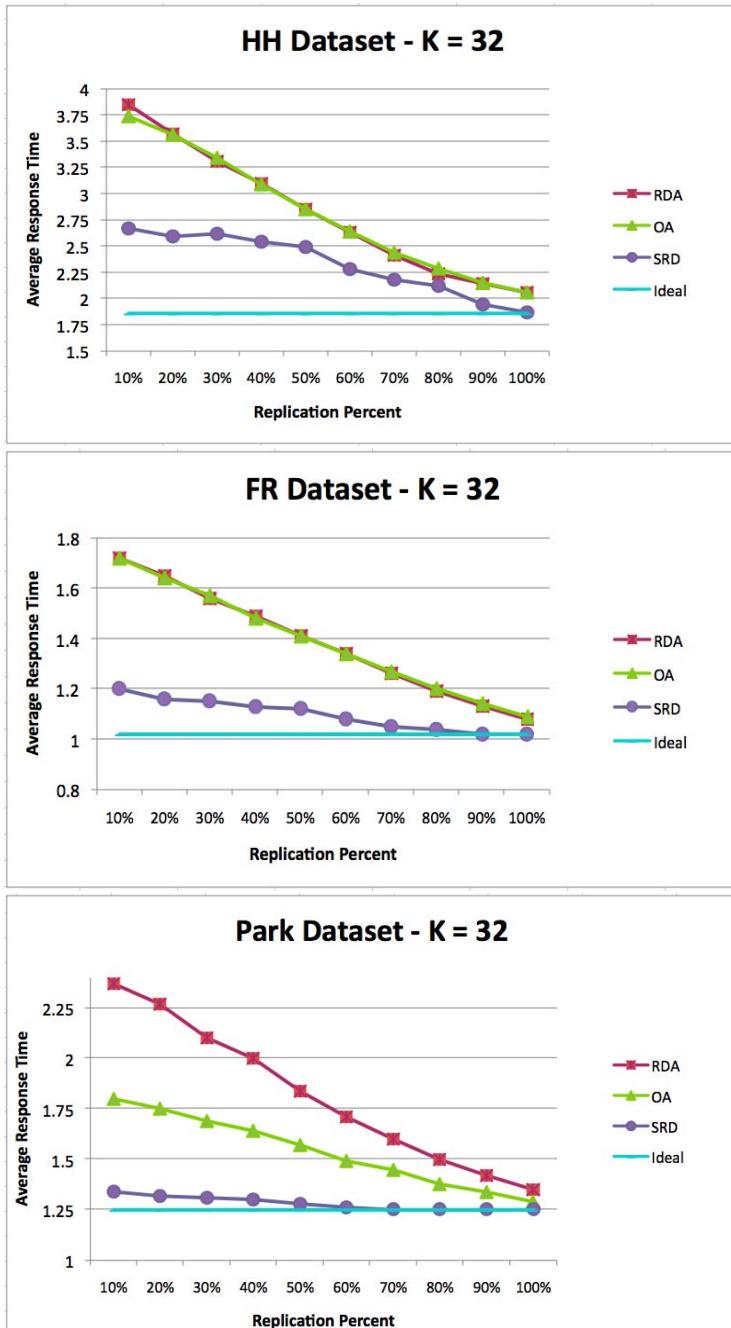


Fig. 2. Comparison of average response time qualities of the Random Duplicate Assignment (RDA), Orthogonal Assignment (OA), and Selective Replicated Declustering (SRD) schemes with increasing replication for $K = 32$ disks

existing algorithms RDA and OA show very close performance for the *HH* and *FR* datasets, whereas OA performs better than RDA for the *Park* dataset.

As seen in Figs. 1 and 2, the proposed SRD algorithm performs better than the RDA and OA algorithms for all declustering instances. We observe that the proposed SRD algorithm achieves very close to the ideal values and in fact achieves ideal results even for replication amounts less than 100% in all datasets apart from 32-way declustering of the *HH* dataset (where it achieves the ideal result at 100% replication). Furthermore, SRD provides very good response time results even for very low replication amounts such as 10% or 20%.

5 Conclusions

We proposed and implemented an efficient and effective iterative improvement heuristic for selective replicated two-way declustering that utilizes a given query distribution and a recursive framework to obtain a multi-way replicated declustering. We tested the performance of our algorithm on three real datasets with synthetically generated arbitrary queries. Our initial implementation indicates that the proposed approach is promising since we obtained favorable performance results compared to two state-of-the-art replicated declustering schemes that performs well in arbitrary queries.

As a future work, we will investigate development and implementation of efficient replica deletion schemes, intelligent schemes that set adaptive maximum allowable replication amounts across the levels of the recursion tree, and a multi-way refinement scheme for iterative improvement of the multi-way replicated declustering obtained through recursive two-way declusterings. This work assumes homogeneous data item sizes and homogeneous disks. Heterogeneity in both aspects can also be considered as a future research area.

References

1. Tosun, A.S.: Threshold-based declustering. *Information Sciences* 177(5), 1309–1331 (2007)
2. Koyuturk, M., Aykanat, C.: Iterative-improvement-based declustering heuristics for multi-disk databases. *Information Systems* 30, 47–70 (2005)
3. Liu, D.R., Shekhar, S.: Partitioning similarity graphs: a framework for declustering problems. *Information Systems* 21, 475–496 (1996)
4. Liu, D.R., Wu, M.Y.: A hypergraph based approach to declustering problems. *Distributed and Parallel Databases* 10(3), 269–288 (2001)
5. Ozdal, M.M., Aykanat, C.: Hypergraph models and algorithms for data-pattern-based clustering. *Data Mining and Knowledge Discovery* 9, 29–57 (2004)
6. Demir, E., Aykanat, C., Cambazoglu, B.B.: A link-based storage scheme for efficient aggregate query processing on clustered road networks. *Information Systems* (2009), doi:10.1016/j.is.2009.03.005
7. Demir, E., Aykanat, C., Cambazoglu, B.B.: Clustering spatial networks for aggregate query processing: A hypergraph approach. *Information Systems* 33(1), 1–17 (2008)

8. Tosun, A.S.: Analysis and comparison of replicated declustering schemes. *IEEE Trans. Parallel Distributed Systems* 18(11), 1587–1591 (2007)
9. Sanders, P., Egner, S., Korst, K.: Fast concurrent access to parallel disks. In: Proc. 11th ACM-SIAM Symp. Discrete Algorithms, pp. 849–858 (2000)
10. Tosun, A.S.: Replicated declustering for arbitrary queries. In: Proc. 19th ACM Symp. Applied Computing, pp. 748–753 (2004)
11. Tosun, A.S.: Design theoretic approach to replicated declustering. In: Proc. Int'l Conf. Information Technology Coding and Computing, pp. 226–231 (2005)
12. Fiduccia, C.M., Mattheyses, R.M.: A linear-time heuristic for improving network partitions. In: Proc. of the 19th ACM/IEEE Design Automation Conference, pp. 175–181 (1982)
13. Chen, L.T., Rotem, D.: Optimal response time retrieval of replicated data. In: Proc. 13th ACM SIGACT-SIGMOD-SIGART symposium on principles of database systems, pp. 36–44 (1994)

Topic 6

Grid, Cluster, and Cloud Computing

Introduction

Jon Weissman*, Lex Wolters*, David Abramson*, and Marty Humphrey*

Grid, Cluster, and Cloud Computing represent important points on the parallel and distributed computing platform landscape. Grids and clusters continue to generate interesting and important research. The inclusion of Cloud computing is new for 2009 and we are happy to report that our track has two high quality papers on Clouds. This year the track was very competitive and 7 out of 27 papers were selected, an acceptance rate that is lower than the conference as a whole. The papers were carefully reviewed by all chairs with 4 reviews per paper and a clear consensus emerged on the top 7 papers. We are pleased that one of our papers (POGGI) was selected as one of the top 4 best papers in the conference. The papers spanned many countries in Europe and Asia and covered a wide-array of topics including programming systems, infrastructure, and applications.

As topic Chairs we wish to acknowledge all those that contributed to this effort and in particular the authors of the submitted papers and the reviewers that contributed their valuable time and expertise to the selection process.

* Topic Chairs.

POGGI: Puzzle-Based Online Games on Grid Infrastructures*

Alexandru Iosup

Electrical Eng., Mathematics and Computer Science Department

Delft University of Technology, Delft, The Netherlands

A.Iosup@tudelft.nl

Abstract. Massively Multiplayer Online Games (MMOGs) currently entertain millions of players daily. To keep these players online and generate revenue, MMOGs are currently relying on manually generated content such as logical challenges (puzzles). Under increased demands for personalized content from a growing community, it has become attractive to generate personalized puzzle game content automatically. In this work we investigate the automated puzzle game content generation for MMOGs on grid infrastructures. First, we characterize the requirements of this novel grid application. With long-term real traces taken from a popular MMOG we show that hundreds of thousands of players are simultaneously online during peak periods, which makes content generation a large-scale compute-intensive problem. Second, we design the POGGI architecture to support this type of application. We assess the performance of our reference implementation in a real environment by running over 200,000 tasks in a pool of over 1,600 nodes, and demonstrate that POGGI can generate commercial-quality content efficiently and robustly.

1 Introduction

Massively Multiplayer Online Games (MMOGs) have emerged in the past decade as a new type of large-scale distributed application: real-time virtual world simulations entertaining at the same time millions of players located around the world. In real deployments, the operation and maintenance of these applications includes two main components, one dealing with running the large-scale simulation, the other with populating it with content that would keep the players engaged (and paying). So far, content generation has been provided exclusively by human content designers, but the growth of the player population, the lack of scalability of the production pipeline, and the increase in the price ratio between human work and computation make this situation undesirable for the future. In this work we formulate and investigate the problem of automated content generation for MMOGs using grids.

* We gratefully thank Dr. Dick Epema and the Delft ICT Talent Grant for support, and Dr. Miron Livny for access to the experimental environment.

MMOGs are increasingly present in people's lives and are becoming an important economic factor. The number of players has increased exponentially over the past ten years [1], to about 25 million players at the end of 2008 [1,2]. Successful MMOGs such as World of Warcraft and Runescape number each over 3,000,000 active players.

There are many types of content present in MMOGs, from 3D objects populating the virtual worlds, to abstract challenges facing the players. The MMOG operators seek from these content types the ones that entertain and challenge players for the longest time, thus leading to revenue [3]. *Puzzle games*, that is, games in which the player is entertained by solving a logical challenge, are an important MMOG content type; for instance, players may spend hours on a chess puzzle [4,5] that enables exiting a labyrinth. Today, puzzle game content is generated by teams of human designers, which poses scalability problems to the production pipeline. While several commercial games have made use of content generation [6,7,8,9], they have all been small-scale games in terms of number of players, and did not consider the generation of puzzle game content. In contrast, in this work we address a new research topic, the large-scale generation of puzzle game instances for MMOGs on grid infrastructures.

Invented and promoted by industry, MMOGs have recently started to attract the interest of the distributed systems [10] and database [11] communities. However, these communities have focused so far on resource management for large-scale MMOGs, spurring a host of results in resource provisioning [2], scalability [12,13], etc. for the world simulation component. In this work we propose the new research direction of generating (puzzle game) content for MMOGs. Towards this end, our contribution is threefold:

1. We formulate the problem of puzzle game generation as a novel large-scale, compute-intensive application different from typical applications from grids and parallel production environments (Section 2);
2. We propose an application model (Section 3) and an architecture (Section 4) for generating puzzle games at large scale;
3. We show through experiments in a real environment that our proposed architecture can generate content of commercial quality (Section 5).

2 Problem Formulation

In this section we formulate the problem of generating puzzle content for MMOGs.

2.1 The Need for Automatic Puzzle Content Generation

Puzzle games have two attractive characteristics: they can be embedded in MMOGs such as World of Warcraft to keep online the players who prefer thinking over repetitive activities, and they are preferred over other genres by the majority of the online game players [14, p.24]. However, the current industry approach for puzzle content generation does not scale. Due to the business model associated with MMOGs, in which the content is the main factor in attracting

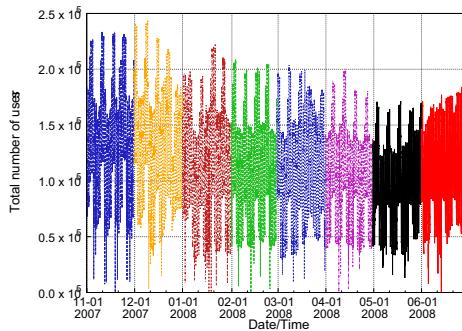


Fig. 1. The number of Active Concurrent Players in the RuneScape MMOG between Nov 1, 2007 and Jul 1, 2008. Each month is depicted with a different type of line.

and retaining paying players [3], game operators prefer with few exceptions to generate the content in-house. Thus, the current market practice is to employ large teams of (human) designers for game content generation. On major titles, teams of 50 people or more can be involved with the content creation pipeline at the same time; for increased scalability such teams are broken down into production units of several persons each under a production hierarchy, which raises the production cost. The manual content generation approach does not scale to larger teams, due to cost and time overheads. This problem is made more difficult by the impossibility to know in advance how the player population will evolve over time: for some MMOGs the population will ramp up or scale down in a matter of weeks or even days [2].

2.2 Challenges in Puzzle Game Content Generation for MMOGs

We identify two main challenges in solving the problem of puzzle content generation for MMOGs: puzzle difficulty balance and scalability to large numbers of players. Failing to meet any of these challenges has a direct impact on revenues: it leads to a degraded game experience and causes the players to leave the game [3][15].

The MMOG players gain virtual reputation with their ability to excel in game activities such as puzzle solving. Thus, the puzzle game instances that various players solve must be matched with the player ability, and be comparable in difficulty for players of the same ability; we call this combined challenge the *puzzle difficulty balance*.

The ability to scale to the large numbers of players specific to MMOGs means that enough puzzle game instances are available for each player requiring them, so that the response time after requesting an instance is very low; we call this the *scalability* challenge. While popular MMOGs may have millions of users, only a fraction are active at the same time, the Active Concurrent Players (ACP) [3]. Thus, the number of puzzle game instances needed at any time is limited by the peak ACP volume. To understand the ACP variation over time we have collected

Table 1. Comparison between the Puzzle Generation workloads and the workloads of grid and parallel production environments (PPEs)

	Workload Type		
	Puzzle Gen.	Grids [16]	PPEs [17]
Users	100,000s	10s-100s	10s-100s
Performance metrics	HTC and HPC	HPC [18]	HPC [19]
Workload	Very Dynamic	Dynamic	Static
Response Time	Very Low	High	Very High
# of Jobs/Result	10s-1,000s	10s-100,000s	1-100s
Task coupling	Workflows	Workflows, Bags-of-Tasks	Parallel, Single

traces from RuneScape, the second-most popular MMOG by number of active players. The operators of RuneScape publish official ACP counts through a web service interface; only the ACP count for the last few seconds is available. We have collected over 8 months of ACP count samples with a 2-minute sampling interval. Figure 11 shows that for RuneScape the daily ACP peaks are between 100,000 and 250,000 players world-wide. (Peaks are several hours long.)

2.3 Puzzle Content Generation Using Grid Infrastructures

Our vision is that the puzzle game content will be automatically generated by a computing environment that can dynamically adjust to a large-scale, compute-intensive, and highly variable workload. Grid environments and to some extent even today's parallel production environments (PPEs) match well the description of this environment. However, whether these environments can support the MMOG puzzle generation workloads and their associated challenges is an open research question. Table 11 compares the characteristics of the MMOG workloads with those of the typical workloads present in these environments. Both the current grids and PPEs serve two-three orders of magnitude fewer users than needed for MMOGs. Grids and PPEs handle well dynamic and static workloads, but have problems dealing with the very dynamic workload characteristics of MMOGs (such as bursts of jobs). The response time model for grid and PPE applications permits middleware overheads higher than what is acceptable for MMOG workloads. Efficiently running in grids large workflows or large numbers of related jobs to produce a unique final result are active research topics [20,21].

For the moment, we do not see as viable an approach in which the machines of the players are used to generate content without some form of supervision or verification. Though the machines of the active players may have enough spare cycles, as in MMOGs players compete with each other for virtual or even real-world gains, cheating can be financially profitable. Moreover, the game developers do not want to give away their content generation algorithms, which are an important competitive advantage, even in binary form. More work on distributed trust and digital rights management is needed before such an approach can be viable.

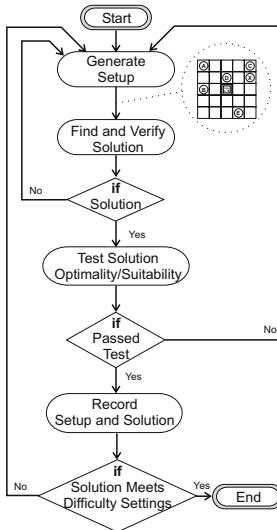


Fig. 2. The workflow structure of a generic Puzzle game instance generator

3 Application Model

In this section we model puzzle content generation as a workflow; ours is the first workflow formulation for this novel application domain.

The puzzle game generation application consists of two main functional phases: generating a solvable puzzle game instance and finding a solution for it, and testing that the proposed solution is minimal in solving effort. The second phase addresses the puzzle difficulty balance challenge (see Section 2.2), so that players cannot solve a generated puzzle in a simpler way. The two functional phases are executed until both complete successfully, or until the time allocated for their execution is exceeded. Thus, our puzzle generation application is one of the first iterative grid workflows; given the number of iterations observed in practice (see Section 5.2), it is also one of the largest.

3.1 Workflow Structure

We model the generic puzzle game generation application as the workflow with seven levels depicted in Figure 2: *Generate Setup*, *Find and Verify Solution*, *if Solution*, *Test Solution Optimality/Suitability*, *if Passed Test*, *Record Setup and Solution*, and *if Solution Meets Difficulty Settings*. The first three (the following two) levels correspond to the first (second) functional phase of the workflow; the other levels are added for output and ending purposes.

The *Generate Setup* level generates a puzzle game setup, such as a board and the pieces on it, that needs to be solved. The execution time of this level depends on what functionality it provides: this level may be designed to generate random setups including ones that break the rules of the puzzles, or include the verification of the setup validity against the rules.

The *Find and Verify Solution* level solves the puzzle starting from the setup generated in the previous level and using one of the many game solving techniques to evolve from setup to the solution state; for reviews of these techniques we refer to [22, 23, 24]. Depending on the amenability of the puzzle to different solving strategies, this level may employ algorithms from brute force search to genetic algorithms, game-tree search, and SAT-solving. Regardless of the algorithm employed, the execution time for this level is bounded by the maximum execution time set by the application designer.

The *Test Solution Optimality / Suitability* level attempts to determine if the found solution is optimal, and if it is suitable for being recorded (i.e., is the solution of the right complexity, but also of the type that will entertain the players?). Depending on the difficulty of testing these two aspects of the solution, the execution time of this level may be similar to that of the *Find and Verify Solution* level.

The *if Solution Meets Difficulty Settings* level was introduced at the end of the workflow to ensure the recording of puzzle setups whose best solutions do not meet the current difficulty settings, but have passed all the other tests and may therefore be used for lower difficulty settings.

3.2 Example: The Lunar Lockout Puzzle

We demonstrate the use of the application model introduced in the previous section by applying it to a sample puzzle game generation application: the generation of instances for the Lunar Lockout puzzle [25]. This commercial puzzle is played on a board on which pins are placed from the start and may be moved according to the following game rules to a goal position. A move consists of pushing a pin either horizontally or vertically, until it is blocked by another pin. The moved pin will be placed in the board cell just before the cell of the blocking pin, considering the direction of the push. The moved pin cannot hop over another pin or be moved outside the board. The goal is to place the X pin on a target position. A solution consists of a time-ordered set of pin movements leaving the 'X' pin on the target position. The puzzle is characterized by the size of the board N , the number of pins P , and the difficulty settings D (i.e., number of moves for an entertaining solution depending on the player's level). A *puzzle setup* consists of N , P , and the initial position of the pins; see Section 5.1 for a generated puzzle setup. The mapping of this application to a puzzle-generating workflow is described below.

The *Generate Setup* level generates a random positioning of the pins on the board, ensuring that the basic rules of the game are enforced, e.g., the pins do not overlap.

The *Find and Verify Solution* level uses backtracking to solve the puzzle. This corresponds to the situation common in practice where no faster solving algorithms are known. The choice of backtracking also ensures low memory consumption and efficient cache usage for today's processors. This workflow level stops when a solution was found.

The *Test Solution Optimality / Suitability* level also applies backtracking to find all the other solutions with a lower or equal number of movements (size). This level needs to investigate moves that were not checked by the *Find and Verify Solution* level; for backtracking this can be easily achieved with minimal memory consumption and without redundant computation. If a solution with a lower number of movements is found, it becomes the new solution and the process continues. The solution suitability test verifies that the best found solution has a minimum size that makes the puzzle interesting for its players, e.g., the size of the solution is at least 4 for beginner players.

4 The POGGI Architecture

In this section we present the Puzzle-Based Online Games on Grid Infrastructures (POGGI) architecture for puzzle game generation on grid infrastructures.

4.1 Overview

The main goal of POGGI is to meet the challenges introduced in Section 2.2. In particular, POGGI is designed to generate puzzle game content efficiently and at the scale required by MMOGs by executing large numbers of puzzle generation workflows on remote resources. We focus on three main issues:

1. *Reduce execution overheads.* First, by not throwing away generated content that does not meet the current but may meet future difficulty settings (see the last workflow level in Section 3.1) our architecture efficiently produces in advance content for future needs. Second, in previous work [26] we have found the job execution overhead to be an important performance bottleneck of current grid workflow engines; for this reason, no current (grid) workflow engine can be used to handle the large number of tasks required by our application (see Section 5.2). To address this problem, we have developed a workflow engine dedicated to puzzle content generation.

2. *Adapt to workload variability.* Using the long-term traces we have acquired from one of the Top-5 MMOGs, we have recently shown [2] that MMOGs exhibit high resource demand variability driven by a user presence pattern that changes with the season and is also fashion-driven, and by a user interaction pattern that changes with the gameplay style. We use detailed statistics of the puzzle use and of the content generation performance to adapt to workload variability.

3. *Use existing middleware.* The architecture uses an external Resource Management Service (RMS), e.g., Condor [27], Globus and Falkon [28], to execute reliably bags-of-tasks in which each task is a puzzle generation workflow, and to monitor the remote execution environment. The architecture is also interfacing with an external component, *Game Content*, which stores the generated content, and which collects statistical data about the actual usage of this content.

4.2 Main Components

The six main components of POGGI are depicted in Figure 3, with rectangles representing (nested) components, and arrows depicting control and data flows.

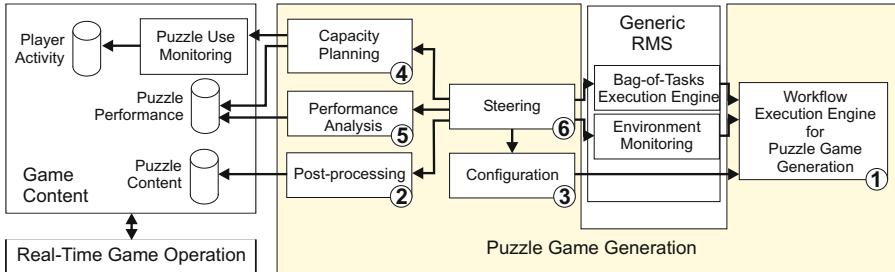


Fig. 3. Overview of the POGGI architecture for puzzle game content generation

1. Workflow Execution Engine for Puzzle Game Generation and 2. Post-Processing: Generic workflow execution engines that can run the puzzle generation workflows (see Section 3.1) already exist [29, 30, 31, 32]. However, they are built to run the workflow tasks on remote resources through the services of the RMS, which leads to high communication and state management overheads [28, 26]. In contrast to this approach, we introduce in our architecture a specialized component for the execution of the puzzle generation workflows on a single resource; we show in Section 5 that our workflow engine can handle the large number of tasks required for puzzle content generation. While this component can be extended to execute workflow tasks on multiple resources similarly to the execution of bag-of-tasks by Falkon [28], the presence of thousands of other workflow instances already leads to high parallelism with minimal execution complexity. The *Post-Processing* component parses the workflow output and stores the generated content into a database.

3. Configuration: The generic workflow model introduced in Section 3.1 can generate content for all the difficulty levels, if the difficulty settings are set to the maximum and the suitability level is set to the minimum. Then, all the generated setups that are solvable are recorded. However, this configuration is inefficient in that high difficulty states will be explored even for beginner players. Conversely, settings that are too strict may lead to ignoring many possibly useful setups. Thus, we introduce in the architecture a component to set the appropriate configuration depending on the level of the players in need of new content.

4. Capacity Planning: As shown in Figure 1, the ACP volume reaches daily peaks of twice the long-term average. In addition, the use of a specific puzzle game may be subject to seasonal and even shorter-term changes. Thus, it would be inefficient to generate content at constant rate. Instead, the capacity planning component analyzes the use of puzzles and the performance of the content generation process and gives recommendations of the needed number of resources. Given the high number of players, enough performance analysis data is present almost from the system start-up in the historical records. We have evaluated various on-line prediction algorithms for MMOG capacity planning in our previous work [2].

5. Performance Analysis: Detailed performance analysis is required to enable the capacity planning component. The performance analysis component focuses

on two important performance analysis aspects: extracting metrics at different levels, and reporting more than just the basic statistics. We consider for performance analysis job-, operational-, application-, and service-level metrics. The first two levels comprise the traditional metrics that describe the execution of individual [19] and workflow [33] jobs. The *application-level metrics* and the *service-level* metrics are specific to the puzzle generation application. At the application-level the analysis follows the evolution of internal application counters (e.g., number of states explored) over time. At the service-level the analysis follows the generation of interesting puzzle game instances. The performance analysis component performs an in-depth statistical analysis of metrics at all levels.

6. *Steering*: To generate puzzle game instances, the various components of our architecture need to operate in concert. The steering component triggers the execution of each other component, and forcefully terminates the puzzle generation workflows that exceed their allotted execution time. Based on the capacity planning recommendations and using the services of the generic RMS, it executes the puzzle generation workflows.

4.3 Implementation Details

We have implemented the Steering component of our architecture on top of the GrenchMark [34, 26] grid testing tool, which can already generate and submit multi-job workloads to common grid and cluster resource management middleware such as Condor, Globus, SGE, and PBS. Thus, the POGGI architecture is not limited to a single middleware, and can already operate in many deployed environments. For this work we have extended the workload generation and management features of GRENCHMARK, in particular with the ability to generate the bags-of-tasks comprising puzzle generation workflows.

We have built the Performance Analysis component on top of the GrenchMark tool for analyzing workload execution, which can already extract performance metrics at job and operational levels [26]. We have added to this component the ability to extract performance metrics at the application and service levels.

5 Experimental Results

In this section we present our experimental results, which demonstrate that POGGI can be used in real conditions to generate commercial-quality content.

We have performed the experiments in the Condor pool at U.Wisconsin-Madison, which comprises over 1,600 processors. The system was shared with other users; for all experiments we have used a normal priority account.

5.1 Lunar Lockout: Solved and Extended

The commercial version of the game [25] consists of a playing set and cards describing 40 puzzle instances. These instances have been generated manually by a team of three content designers over a period of about one year. The instances can be characterized in our application model (see Section 3.2) as $N = 5$, $P =$

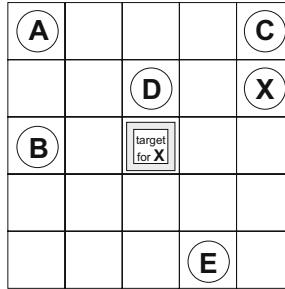


Fig. 4. Lunar Lockout instance

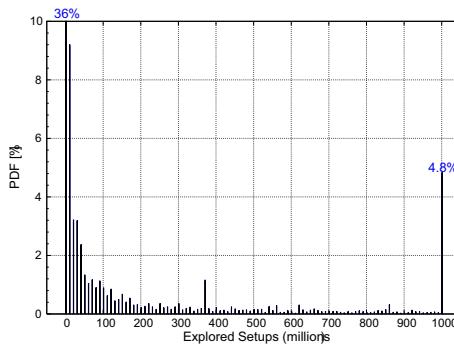


Fig. 5. The PDF of the number of explored puzzle setups per workflow. Each bar represents a range of 10 million puzzle setups.

$4 - 6$, and D such that the solution size ranges from 4 (beginner player) to 10 (advanced). During our experiments we have generated and solved all the boards that come with the commercial version of the game. This demonstrates that our architecture can be used to produce commercial-quality content much faster than by using the manual approach.

In addition, we have generated automatically many new puzzle instances with equal or larger boards ($N > 5$), more pins ($P > 6$), and solutions of up to 21 moves, corresponding to an expert play level that exceeds the human design capabilities. Figure 4 depicts a sample setup that has been automatically generated for $N = 5$, $P = 6$, and $D = \{ \text{solution size} \geq 8 \text{ for advanced players, solution size} \geq 4 \text{ for beginner players,} \dots \}$. It turns out that the best solution for moving the X pin to the target has 8 moves: A→Right, A→Down, X→Left, A→Up, E→Up, A→Left, X→Down, and X→Left.

5.2 Application Characterization

To characterize the puzzle generation application we use a test workload comprising 10,000 workflows. Overall, the execution of this workload led to the

Table 2. The number of interesting states found for each 1,000 jobs, per solution size.

Configuration	Solution Size																
	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Normal	1,281	1,286	645	375	230	134	75	47	27	11	6	1	2	-	-	-	-
Large	-	-	409	334	257	147	171	57	79	83	39	41	24	22	2	3	7

evaluation of 1,364 billion of puzzle setups (tasks). The number of tasks is much larger for puzzle content generation than for the largest scientific workflows; the latter comprise rarely over a million of tasks [20].

The probability distribution function (PDF) of the number of explored puzzle setups for this workload is depicted in Figure 5. The distribution is skewed towards left: most of the executed workflows explore fewer puzzle setups than the average of the whole workload. This indicates that in most cases the workflow termination condition (finding a puzzle setup that matches the difficulty settings) is met faster than the workload average indicates.

5.3 Meeting the Challenges

We have identified in Section 2.2 two main challenges for puzzle content generation, puzzle difficulty and scalability. We now show evidence that the POGGI architecture can meet these challenges.

We first evaluate the impact of the application configuration on finding puzzle instances of different difficulty. Two configurations are considered during this experiment: workflows that explore a normal-sized space (*normal instances*), and workflows that explore a large-sized space (*large instances*). Table 2 shows the number of puzzle instances found for these two configurations. The normal workflow instances find more puzzle instances with solution sizes up to 7. For solution sizes of 8 through 12, both instances behave similarly. For solution sizes of 13 and higher, the large workflow instances become the best choice. Based on similar results and on demand the Capacity Planning component can issue recommendations for efficiently finding unique instances of desired difficulty.

To show evidence of scalability we investigate the average response time and the potential for soft performance guarantees. For the 10,000 workflows test workload described in Section 5.2, the average workflow is computed in around 5 minutes; thus, it is possible to generate content for hundreds of thousands of players on a moderately sized grid infrastructure.

We have also evaluated the application- and the service-level throughput over time. We define the application-level (service-level) throughput as the number of (interesting) puzzle setups explored (found) over the time unit, set here to one second; for convenience, we use the terms states and puzzle setups interchangeably. Figure 6 shows the evolution of application- and service-level throughput over time. The architecture achieves an overall application-level throughput of over 15 million states explored per second, and an overall service-level throughput of over 0.5 interesting states discovered per second. The performance decreases with time

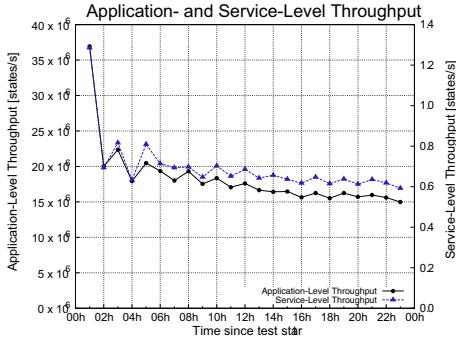


Fig. 6. The evolution of the application- and service-level throughput over time

due to Condor’s fair sharing policy: our normal user’s priority degrades with the increase of resource consumption. The performance decrease becomes predictable after about 6 hours. This allows a service provider to practically guarantee service levels, even in a shared environment. These experiments also demonstrate the ability of our architecture to extract application- and service-level metrics.

6 Related Work

In this section we survey two areas related to our work: game content generation, and many-tasks applications. We have already presented in Section 2.1 the current industry practice.

The topic of automated (procedural) generation has already been approached by the industry as an alternative to manual content generation. In 1984, the single-player game Elite [6] used automated game content generation to simulate a large world on an 8-bit architecture. Since then, several other games have used a similar approach: ADOM [7] generates battle scenes that adapt dynamically to the player level, Diablo [8] generates instances of enclosed areas for the player to explore, The Dwarf Fortress [9] generates an entire world from scratch, etc. All these approaches were games with a small numbers of concurrent players or even with a single player, and generated content on the (main) player’s machine. In contrast, this work focuses on the efficient generation of puzzle game instances for MMOGs, using a large resource pool as the computing infrastructure.

Until a few years ago, few environments existed that could manage the high number of jobs required by MMOGs, among them SETI@Home [35]. More recently, tools such as Falkon [28] and Swift (through Falkon) have started to address the problem of executing with low overhead large numbers of bags-of-tasks and workflows, respectively. In contrast with all these approaches, our architecture optimizes the execution of a specific application (though with a much wider audience) and specifically considers dynamic resource provisioning adjustments to maintain the performance metrics required by application’s commercial focus.

7 Conclusion and Ongoing Work

With a large and growing user base that generates large revenues but also raises numerous technological problems, MMOGs have recently started to attract the interest of the research community. In this work we are the first to identify the problem of the scalability of content generation. To address this problem, we have designed an implemented POGGI, an architecture for automatic and dynamic content generation for MMOGs. Our architecture focuses on puzzle game content generation, which is one of the most important components of the generic game content generation problem. Experimental results in a large resource pool show that our approach can achieve and even exceed the manual generation of commercial content.

Currently, we are extending our architecture with more game content types and with mechanisms for malleable workflow execution. For the future, we plan make POGGI less domain-specific, towards generic scientific computing support.

References

1. Woodcock, B.S.: An analysis of mmog subscription growth. Online Report (2009), <http://www.mmogchart.com>
2. Nae, V., Iosup, A., Podlipnig, S., Prodan, R., Epema, D.H.J., Fahringer, T.: Efficient management of data center resources for massively multiplayer online games. In: ACM/IEEE SuperComputing (2008)
3. Bartle, R.: Designing Virtual Worlds. New Riders Games (2003) ISBN 0131018167
4. Kasparov, G.: Chess Puzzles Book. Everyman Chess (2001)
5. Polgar, L.: Chess: 5334 Problems, Combinations and Games. Leventhal (2006)
6. Braben, D., Bell, I.: "Elite", Acornsoft, 1984 (2009), <http://www.iancbell.clara.net/elite/>
7. Biskup, T.: "ADOM", Free (1994), <http://www.adom.de/> (2009)
8. Schaefer, E., et al.: "Diablo I", Blizzard Entertainment (1997), <http://www.blizzard.com/us/diablo/> (2009)
9. Adams, T.: "Dwarf Fortress", Free (2006), <http://www.bay12games.com/dwarves/> (2009)
10. Neumann, C., Prigent, N., Varvello, M., Suh, K.: Challenges in peer-to-peer gaming. Computer Communication Review 37(1), 79–82 (2007)
11. White, W.M., Koch, C., Gupta, N., Gehrke, J., Demers, A.J.: Database research opportunities in computer games. SIGMOD Record 36(3), 7–13 (2007)
12. White, W.M., Demers, A.J., Koch, C., Gehrke, J., Rajagopalan, R.: Scaling games to epic proportion. In: ACM SIGMOD ICMD, pp. 31–42. ACM, New York (2007)
13. Müller, J., Gorlatch, S.: Rokkatan: scaling an rts game design to the massively multiplayer realm. Computers in Entertainment 4(3) (2006)
14. The Entertainment Software Association, "2008 annual report," Technical Report, <http://www.theesa.com> (November 2008)
15. Fritsch, T., Ritter, H., Schiller, J.H.: The effect of latency and network limitations on mmorpgs: a field study of everquest2. In: NETGAMES. ACM, New York (2005)
16. Iosup, A., Dumitrescu, C., Epema, D.H.J., Li, H., Wolters, L.: How are real grids used? the analysis of four grid traces and its implications. In: GRID, pp. 262–269. IEEE, Los Alamitos (2006)

17. Lublin, U., Feitelson, D.G.: The workload on parallel supercomputers: modeling the characteristics of rigid jobs. *J. PDC* 63(11), 1105–1122 (2003)
18. Iosup, A., Epema, D.H.J., Franke, C., Papaspouli, A., Schley, L., Song, B., Yahyapour, R.: On grid performance evaluation using synthetic workloads. In: Frachtenberg, E., Schwiegelshohn, U. (eds.) *JSSPP 2006. LNCS*, vol. 4376, pp. 232–255. Springer, Heidelberg (2007)
19. Feitelson, D.G., Rudolph, L.: Metrics and benchmarking for parallel job scheduling. In: Feitelson, D.G., Rudolph, L. (eds.) *IPPS-WS 1998, SPDP-WS 1998, and JSSPP 1998. LNCS*, vol. 1459, pp. 1–24. Springer, Heidelberg (1998)
20. Raicu, I., Zhang, Z., Wilde, M., Foster, I., Beckman, P., Iskra, K., Clifford, B.: Toward loosely-coupled programming on petascale systems. In: *ACM/IEEE SuperComputing* (2008)
21. Iosup, A., Sonmez, O.O., Anoep, S., Epema, D.H.J.: The performance of bags-of-tasks in large-scale distributed systems. In: *HPDC*, pp. 97–108 (2008)
22. Conway, J.H.: All games bright and beautiful. *The American Mathematical Monthly* 84(6), 417–434 (1977)
23. Bouzy, B., Cazenave, T.: Computer go: An ai oriented survey. *Artificial Intelligence* 132, 39–103 (2001)
24. Demaine, E.D.: Playing games with algorithms: Algorithmic combinatorial game theory. In: Sgall, J., Pultr, A., Kolman, P. (eds.) *MFCS 2001. LNCS*, vol. 2136, pp. 18–32. Springer, Heidelberg (2001)
25. Yamamoto, H., Yoshigahara, N., Tanaka, G., Uematsu, M., Nelson, H.: Lunar Lockout: a space adventure puzzle, *ThinkFun* (1989)
26. Stratan, C., Iosup, A., Epema, D.H.J.: A performance study of grid workflow engines. In: *GRID*, pp. 25–32. IEEE, Los Alamitos (2008)
27. Thain, D., Tannenbaum, T., Livny, M.: Distributed computing in practice: the condor experience. *Conc.&Comp.: Pract.&Exp.* 17, 323–356 (2005)
28. Raicu, I., Zhao, Y., Dumitrescu, C., Foster, I.T., Wilde, M.: Falkon: a fast and light-weight task execution framework. In: *ACM/IEEE SuperComputing* (2007)
29. Singh, G., Kesselman, C., Deelman, E.: Optimizing grid-based workflow execution. *J. Grid Comput.* 3(3-4), 201–219 (2005)
30. Ludäscher, B., et al.: Scientific workflow management and the Kepler system. *Conc. & Comp.: Pract. & Exp.* 18(10), 1039–1065 (2006)
31. Oinn, T.M., et al.: Taverna: lessons in creating a workflow environment for the life sciences. *Conc. & Comp.: Pract. & Exp.* 18(10), 1067–1100 (2006)
32. von Laszewski, G., Hategan, M.: Workflow concepts of the Java CoG Kit. *J. Grid Comput.* 3(3-4), 239–258 (2005)
33. Truong, H.L., Dustdar, S., Fahringer, T.: Performance metrics and ontologies for grid workflows. *Future Gen. Comp. Syst.* 23(6), 760–772 (2007)
34. Iosup, A., Epema, D.H.J.: GrenchMark: A framework for analyzing, testing, and comparing grids. In: *CCGrid*, pp. 313–320. IEEE, Los Alamitos (2006)
35. Anderson, D.P., Cobb, J., Korpela, E., Lebofsky, M., Werthimer, D.: “SETI@Home”. *Commun. ACM* 45(11), 56–61 (2002)

Enabling High Data Throughput in Desktop Grids through Decentralized Data and Metadata Management: The BlobSeer Approach

Bogdan Nicolae¹, Gabriel Antoniu², and Luc Bouge³

¹ University of Rennes 1, IRISA, Rennes, France

² INRIA, Centre Rennes - Bretagne Atlantique, IRISA, Rennes, France

³ ENS Cachan/Brittany, IRISA, France

Abstract. Whereas traditional Desktop Grids rely on centralized servers for data management, some recent progress has been made to enable distributed, large *input* data, using to peer-to-peer (P2P) protocols and Content Distribution Networks (CDN). We make a step further and propose a generic, yet efficient data storage which enables the use of Desktop Grids for applications with high *output* data requirements, where the access grain and the access patterns may be random. Our solution builds on a blob management service enabling a large number of concurrent clients to efficiently read/write and append huge data that are fragmented and distributed at a large scale. Scalability under heavy concurrency is achieved thanks to an original metadata scheme using a distributed segment tree built on top of a Distributed Hash Table (DHT). The proposed approach has been implemented and its benefits have successfully been demonstrated within our *BlobSeer* prototype on the Grid'5000 testbed.

1 Introduction

During the recent years, Desktop Grids have been extensively investigated as an efficient way to build cheap, large-scale virtual supercomputers by gathering idle resources from a very large number of users. Rather than relying on clusters of workstations belonging to institutions and interconnected through dedicated, high-throughput wide-area interconnect (which is the typical physical infrastructure for Grid Computing), Desktop Grids rely on desktop computers from individual users, interconnected through Internet. Volunteer computing is a form of distributed computing in which Desktop Grids are used for projects of public interest: the infrastructure is built thanks to volunteers that accept to donate their idle resources (cycles, storage) in order to contribute to some public cause. These highly-distributed infrastructures can typically benefit to embarrassingly-parallel, computationally intensive applications, where the workload corresponding to each computation can easily be divided into a very large set of small independent jobs that can be scattered across the available computing nodes. In a typical (and widely used) setting, Desktop Grids rely on a master/worker scheme: a central server (playing the master role) is in charge of distributing the small jobs to many volunteer workers that have announced their availability. Once the computations are complete, the results are typically gathered on the central (master) server, which

validates them, then builds a global solution. BOINC [1], XtremWeb [2] or Entropia [3] are examples of such Desktop Grid systems.

The initial, widely-spread usage of Desktop Grids for parallel applications consisting in non-communicating tasks with small input/output parameters is a direct consequence of the physical infrastructure (volatile nodes, low bandwidth), unsuitable for communication-intensive parallel applications with high input or output requirements. However, the increasing popularity of volunteer computing projects has progressively lead to attempts to enlarge the set of application classes that might benefit of Desktop Grid infrastructures. If we consider distributed applications where tasks need very large input data, it is no longer feasible to rely on classic centralized server-based Desktop Grid architectures, where the input data was typically embedded in the job description and sent to workers: such a strategy could lead to significant bottlenecks as the central server gets overwhelmed by download requests. To cope with such data-intensive applications, alternative approaches have been proposed, with the goal of offloading the transfer of the input data from the central servers to the other nodes participating to the system, with potentially under-used bandwidth.

Two approaches follow this idea. One of them adopts a P2P strategy, where the input data gets spread across the distributed Desktop Grid (on the same physical resources that serve as workers) [4]. A central data server is used as an initial data source, from which data is first distributed at a large scale. The workers can then download their input data from each other when needed, using for instance a BitTorrent-like mechanism. An alternative approach [4] proposes to use Content Distribution Networks (CDN) to improve the available download bandwidth by redirecting the requests for input data from the central data server to some appropriate surrogate data server, based on a global scheduling strategy able to take into account criteria such as locality or load balancing. The CDN approach is more costly than the P2P approach (as it relies on a set of data servers), however it is potentially more reliable (as the surrogate data servers are supposed to be stable enough).

In this paper, we make a step further and consider using Desktop Grids for distributed applications with high *output* data requirements. We assume that each such application consists of a set of distributed tasks that *produce and potentially modify large amounts of data* in parallel, under heavy concurrency conditions. Such characteristics are featured by 3D rendering applications, or massive data processing applications that produce data transformations. In such a context, a new approach to data management is necessary, in order to cope with both input and output data in a scalable fashion. Very recent efforts have partially addressed the above issues from the perspective of the *checkpoint* problem: the *stdchk* system [5] proposes to use the Desktop Grid infrastructure to store the (potentially large) checkpoints generated by Desktop Grid applications. This proposal is however specifically optimized for checkpointing, where large data units need to be written sequentially. It relies on a centralized metadata management scheme which becomes a potential bottleneck when data access concurrency is high. Related work has been carried out in the area of parallel and distributed file systems [6][7][8] and archiving systems [9]: in all these systems the metadata management is centralized. We propose a *generic*, yet *efficient* storage solution allowing Desktop Grids to be used by applications that generate large amounts of data (not only for checkpointing!), with

random access patterns, variable access grains, under potentially heavy concurrency. Our solution is based on BlobSeer, a blob management service we developed in order to address the issues mentioned above.

The main contribution of this paper is to demonstrate how the decentralized metadata scheme used in BlobSeer fits the needs of the considered application class, and to demonstrate our claims through extensive experimental evaluations. The paper is structured as follows. Section 2 gives an overview of our approach. Section 3 introduces the proposed architecture, with a focus on metadata management in Section 4. Extensive experimental evaluation is performed in Section 5. On-going and future work is discussed in Section 6.

2 Towards a Decentralized Architecture for Data and Metadata Management in Desktop Grids

A Sample Scenario: 3D Rendering. High resolution 3D rendering is known to be costly. Luckily, as rendering is embarrassingly parallel, both industry [10] and the research community [11] have an active interest in building render farms based on the Desktop Grid paradigm. In this context, one of the limitations encountered regards data management: both rendering input and output data reach huge sizes and are accessed in a fine-grain manner (frame-by-frame) under high contention (task-per-frame). From a more general perspective, we aim at providing efficient data management support on Desktop Grids for workloads characterized by: *large input data, large output data, random access patterns, fine-grain data access for both reading and writing and high access concurrency.*

Requirements for Data Management. Given the workload characterization given above, we can infer the following desirable properties:

Storage capacity and space efficiency. The data storage system should be able to store huge pieces of individual data blobs, for a large amount of data overall. Besides, as we assume the applications to be write-intensive, they may generate many versions of the data. Therefore, a space-efficient storage is highly desirable.

Read and write throughput under heavy concurrency. The data storage system should provide efficient and scalable support for input/output data accesses, as a large number of concurrent processes concurrently potentially read and write the data.

Efficient fine-grain, random access. As the grain and the access pattern may be random, the storage system should rely on a generic mechanism enabling efficient random fine-grain accesses within huge data blobs. We definitely favor this approach rather than relying on optimized sequential writes (as in [5]) to a large number of small files, for manageability reasons.

Versioning support. Providing versioning support is also a desirable feature, favoring concurrency, as some version of a data blob may be read while a new version is concurrently created. This fits the needs of checkpointing applications [5], but is also suitable for efficient pipelining through a sequence of transformations on huge amounts of data.

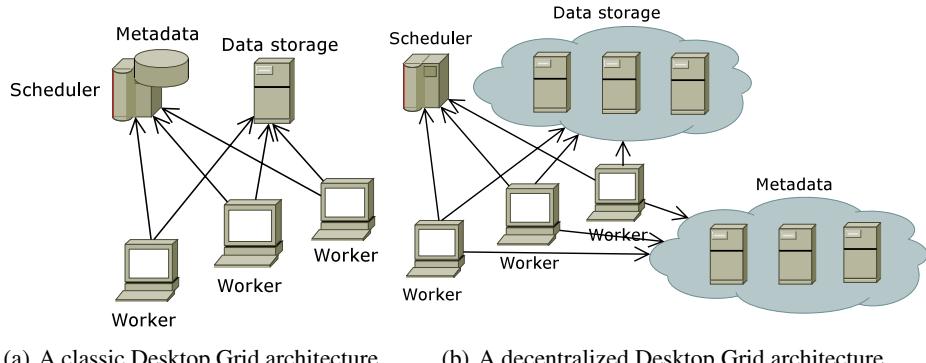


Fig. 1. Metadata and data management in Desktop Grids: centralized versus decentralized

Global Architecture Overview. In a classic Desktop Grid architecture (Figure 1(a)), the workers directly interact with the central scheduler, in order to request a job. Each job is defined by its input/output data, each of which can be downloaded/uploaded from a (typically centralized) location provided by the scheduler. This scheme has several major disadvantages. First, data is stored in a centralized way: this may lead to a significant potential bottleneck for data intensive applications. Second, the whole *metadata* is managed by the central scheduler, which is put under high I/O pressure by all clients that concurrently read the metadata and may potentially become an I/O bottleneck.

Whereas several recent approaches have proposed to decentralize data management in Desktop Grids through P2P or CDN-based strategies [4], to the best of our knowledge, none has explored the idea of decentralizing metadata. We specifically take this approach and propose an architecture based on the BlobSeer prototype, which implements an efficient lock-free, distributed metadata management scheme, in order to provide support for performant concurrent write accesses to data. The Desktop Grid architecture is modified as illustrated on Figure 1(b). The scheduler keeps only a minimal input/output metadata information: the bulk of metadata is delegated to a set of distributed metadata providers. An overview of our approach is given in Section 3, with a focus on metadata management in Section 4. Note that a full description of the algorithms used is available in [12]: in this paper we briefly remind them, then we focus on the impact of distributing data and metadata.

3 The Core of Our Approach: The BlobSeer Service

BlobSeer at a Glance. We have experimentally studied the approach outlined above using our *BlobSeer* prototype: a blob (Binary Large OBject) management service [13][14]. To cope with very large data blobs, BlobSeer uses striping: each blob is made up of blocks of a fixed size $psize$, referred to as *pages*. These pages are distributed among storage space providers. *Metadata* facilitates access to a range ($offset, size$) for any existing version of a blob snapshot, by associating such a range with the physical nodes where the corresponding pages are located.

BlobSeer's client applications may update blobs by writing a specific range within the blob (WRITE). Rather than updating the current pages, each such operation generates a *new* set of pages corresponding to the offset and size requested to be updated. Clients may also APPEND new data to existing blobs, which also results in the creation of new pages. In both cases, metadata is then generated and "weaved" together with the old metadata in such way as to create the illusion of a new incremental snapshot that actually shares the unmodified pages of the blob with the older versions. Thus, two successive snapshots v and $v + 1$ physically share the pages that fall outside of the range of the update that generated snapshot $v + 1$. Metadata is also partially shared across successive versions, as further explained below. Clients may access a specific version of a given blob through the READ primitive.

Metadata is organized as a segment-tree like structure (see Section 4) and is scattered across the system using a Distributed Hash Table (DHT). Distributing data and metadata is the key choice in our design: it enables high performance through parallel, direct access I/O paths, as demonstrated in Section 5.

BlobSeer's Architecture. Our storage infrastructure consists of a set of distributed processes.

Clients may CREATE blobs, then READ, WRITE and APPEND data to them. There may be multiple concurrent clients, and their number may dynamically vary in time.

Data providers physically store the pages generated by WRITE and APPEND. New data providers may dynamically join and leave the system. In a Desktop Grid setting, the same physical nodes which act as workers may also serve as data providers.

The provider manager keeps information about the available storage space and schedules the placement of newly generated pages according to a load balancing strategy.

Metadata providers physically store the metadata allowing clients to find the pages corresponding to the blob snapshot version. We use a distributed metadata management scheme to enhance data throughput through parallel I/O: this aspect is addressed in detail in Section 4. In a Desktop Grid setting, as the data providers, metadata providers can be physically mapped to the physical nodes acting as workers.

The version manager is the key actor of the system. It registers update requests (APPEND and WRITE), assigns snapshot version numbers, and eventually publishes new blob versions, while guaranteeing total ordering and atomicity. In a Desktop Grid setting, this entity can be collocated with the centralized scheduler.

Reading Data. To read data, clients need to provide a blob id, a specific version of that blob, and a range, specified by an offset and a size. The client first contacts the version manager. If the version has been published, the client then queries the metadata providers for the metadata indicating on which providers are stored the pages corresponding to the required blob range. Finally, the client fetches the pages in parallel from the data providers. If the range is not page-aligned, the client may request only the required part of the page from the page provider.

Writing and Appending Data. To write data, the client first determines the number of pages that cover the range to be written. It then contacts the provider manager and

requests a list of page providers able to store the pages. For each page in parallel, the client generates a globally unique page id, contacts the corresponding page provider and stores the contents of the page on it. After successful completion of this stage, the client contacts the version manager to registers its update. The version manager assigns to this update a new snapshot version v and communicates it to the client, which then generates new metadata and “weaves” it together with the old metadata such that the new snapshot v appears as a standalone entity (details are provided in Section 4). Finally, the client notifies the version manager of success, and returns successfully to the user. At this point, the version manager is responsible for eventually publishing the version v of the blob. The APPEND operation is almost identical to the WRITE: the implicit offset is the size of the previously published snapshot version.

4 Zoom on Metadata Management

Metadata stores information about the pages which make up a given blob, for each generated snapshot version. To efficiently build a full view of the new snapshot of the blob each time an update occurs, BlobSeer creates new metadata, rather than updating old metadata. As we will explain below, this decision significantly helps us provide support for heavy concurrency, as it favors *independent concurrent accesses to metadata without synchronization*.

The distributed metadata tree. We organize metadata as a *distributed segment tree* [15]: one such tree is associated to each snapshot version of a given blob id . A segment tree is a binary tree in which each node is associated to a range of the blob, delimited by *offset* and *size*. We say that the node *covers* the range $(offset, size)$. For each node that is not a leaf, the left child covers the first half of the range, and the right child covers the second half. Each leaf covers a single page. We assume the page size $psize$ is a power of two. Figure 2(a) depicts the structure of the metadata for a blob consisting of four pages. We assume the page size is 1. The root of the tree covers the range $(0, 4)$, while each leaf covers exactly one page in this range.

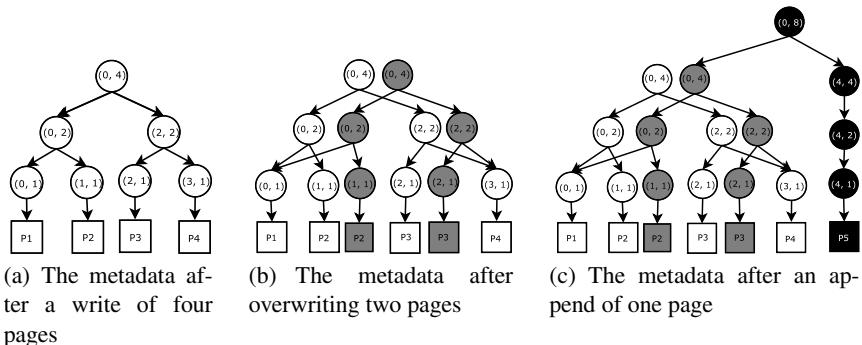


Fig. 2. Metadata representation

To favor efficient concurrent access to metadata, tree nodes are stored on the metadata providers in a distributed way, using a simple DHT (Distributed Hash Table). Each tree node is identified uniquely by its version and range specified by the offset and size it covers.

Sharing metadata across snapshot versions. Such a metadata tree is created when the first pages of the blob are written, for the range covered by those pages. To avoid the overhead (in time and space!) of rebuilding such a tree for the subsequent updates, we create new tree nodes only for the ranges that *do* intersect with the range of the update. These new tree nodes are “weaved” with existing tree nodes generated by past updates (for ranges that do not intersect with the range of the update), in order to build a new consistent view of the blob, corresponding to a new snapshot version. Figure 2(b) shows how metadata evolves when pages 2 and 3 of the 4-page blob represented on Figure 2(a) are modified for the first time.

Expanding the metadata tree. APPEND operations make the blob “grow”: consequently, the metadata tree gets expanded, as illustrated on Figure 2(c), where new metadata tree nodes are generated, to take into account the creation of a fifth page by an APPEND operation.

Reading metadata. Metadata is accessed during a READ operation in order to find out what pages fully cover the requested range R . This involves traversing down the segment tree, starting from the root that corresponds to the requested snapshot version. To start the traversal, the client gets the root from the version manager, which is responsible to store the mapping between all snapshot versions and their corresponding roots. A node N that covers segment R_N is explored if the intersection of R_N with R is not empty. Since tree nodes are stored in a DHT, exploring a node involves fetching it from the DHT. All explored leaves reached this way provide information allowing to fetch the contents of the pages from the data providers.

Writing metadata. For each update (WRITE or APPEND) producing a snapshot version v , it is necessary to build a new metadata tree (possibly sharing nodes with the trees corresponding to previous snapshot versions). This new tree is the smallest (possibly incomplete) binary tree such that its leaves are exactly the leaves covering the pages of range that is written. The tree is built bottom-up, from the leaves towards the root. Note that inner nodes may have children which do *not* intersect the range of the update to be processed. For any given snapshot version v , these nodes form the *set of border nodes* B_v . When building the metadata tree, the versions of these nodes are required. The version manager is responsible to compute the versions of the border nodes such as to assure proper update ordering and consistency.

5 Experimental Evaluation

To illustrate the advantages of our proposed approach we have performed a set of experiments that study the impact of both data and metadata distribution on the achieved

performance levels under heavy write concurrency. All experimentation has been performed using our BlobSeer prototype.

Our implementation of the metadata provider relies on a custom DHT (Distributed Hash Table) based on a simple static distribution scheme. Communication between nodes is implemented on top of an asynchronous RPC library we developed on top of the Boost C++ ASIO library [16]. The experiments have been performed on the Grid'5000 [17] testbed, a reconfigurable, controllable and monitorable Grid platform gathering 9 sites in France. For each experiment, we used nodes located within a single site (Rennes or Orsay). The nodes are outfitted with x86_64 CPUs and 4 GB of RAM. Intracluster bandwidth is 1 Gbit/s (measured: 117.5MB/s for TCP sockets with MTU = 1500 B), latency is 0.1 ms.

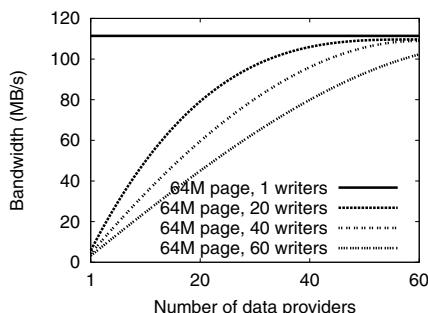
5.1 Benefits of Data Decentralization

To evaluate the impact of data decentralization on the workers' write performance, we consider a set of concurrent workers that write the output data in parallel and we measure the average write bandwidth.

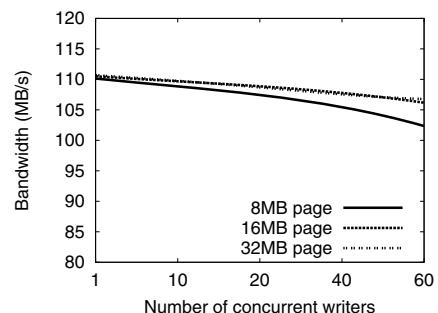
Impact of the Number of Data Providers. In this setting, we deploy a version manager, a provider manager and a variable number of data providers. We also deploy a fixed number of workers and synchronize them to start writing output data simultaneously. Each process is deployed on a dedicated node within the same cluster. As this experiment aims at evaluating the impact of *data* decentralization, we fix the page size at 64 MB, large enough as to generate only a minimal metadata management overhead. We assume each worker writes a single page. A single metadata provider is thus sufficient for this experiment.

Each worker generates and writes its output data 50 times. We compute the average bandwidth achieved by all workers, for all their iterations. The experiment is repeated by varying the total number of workers from 1 to 60.

Results are shown on Figure 3(a): for one single worker, using more than one data provider does not make any difference since a single data provider is contacted at the



(a) Impact of data distribution on the average output bandwidth of workers



(b) Impact of job output size on the average output bandwidth of workers

Fig. 3. Data distribution benefits under high write concurrency

same time. However, when multiple workers concurrently write their output data, the benefits of data distribution become visible. Increasing the number of data providers leads to a dramatic increase in bandwidth performance: from a couple of MB/s to over 100 MB/s when using 60 data providers. Bandwidth performance flattens rapidly when the number of data providers is at least the number of workers. This is explained by the fact that using at least as many data providers as workers enables the provider manager to direct each concurrent write request to a distinct data provider. Under such conditions, the bandwidth measured for a single worker under no concurrency (115 MB/s) is just by 12% higher than the average bandwidth reached when 60 workers write the output data concurrently (102 MB/s).

Impact of the Page Size. We then evaluate the impact of the data output size on the achieved write bandwidth. As in the previous setting, we deploy a version manager, a provider manager and a metadata provider. This time we fix the number of providers to 60. We deploy a variable number of workers and synchronize them to start writing output data simultaneously. Each worker iteratively generates its job output and writes it as a single page to BlobSeer (50 iterations). The achieved bandwidth is averaged for all workers. We repeat the experiment for different sizes of the job output: 32 MB, 16 MB, 8 MB. As can be observed in Figure 3(b), a high bandwidth is sustained as long as the page is large enough. The average client bandwidth drops from 110 MB/s (for 32 MB pages) to 102 MB/s (for 8 MB pages).

5.2 Benefits of *Metadata Decentralization*

In the previous set of experiments we have intentionally minimized the impact of metadata management, in order to emphasize the impact of data distribution. As a next step, we study how metadata decentralization impacts performance. We consider a setting with a large number of workers, each of which concurrently generates many pages, so that metadata management becomes a concern.

Impact of the Number of Metadata Providers. To evaluate the impact of metadata distribution as accurately as possible, we first deploy as many data providers as workers, to avoid potential bottlenecks on the providers. Each process is deployed on a separate physical node. We deploy a version manager, a provider manager and a fixed number of 60 providers. We then launch 60 workers and synchronize them to start writing output data simultaneously. Each worker iteratively generates a fixed-sized 64 MB output and writes it to BlobSeer (50 iterations). The achieved bandwidth is averaged for all workers. We vary the number of metadata providers from 1 to 30. We repeat the whole experiment for various page sizes (64 KB, 128 KB and 256 KB).

Results on Figure 4(a) show that increasing the number of metadata providers results in an improved average bandwidth under heavy concurrency. The improvement is more significant when reducing the page size: since the amount of the associated metadata doubles when the page size halves, the I/O pressure on the metadata providers doubles too. We can thus observe that the use of a centralized metadata provider leads to a clear bottleneck (62 MB/s only), whereas using 30 metadata providers improves the write bandwidth by over 20% (75 MB/s).

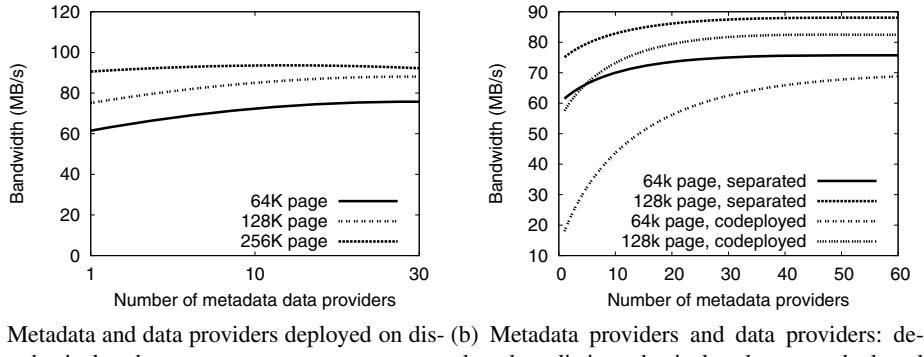


Fig. 4. Metadata striping benefits under high write concurrency

Impact of the Co-Deployment of Data and Metadata Providers. In order to increase the scope of our evaluation without increasing the number of physical network nodes, we perform an additional experiment. We keep exactly the same setting as previously, but we co-deploy a data provider and a metadata provider on each physical node (instead of deploying them on separate nodes). We expect to measure a consistent performance drop and establish a correlation with the results of the previous experiment. This can enable us to predict the performance behavior of our system for larger physical configurations.

Results are shown in Figure 4(b), for two page sizes: 128 KB and 64 KB. For reference, the results obtained in the previous experiment for the same job outputs are plotted on the same figure. We observe a 11% decrease for a 64 KB page size and 7% decrease for a 128 KB page size when using 60 metadata providers. Notice the strong impact of the co-deployment when using a single metadata provider. In this case the I/O pressure on the metadata provider adds to the already high I/O pressure on the co-deployed provider, bringing the bandwidth drop to more than 66%.

The Torture Test: Putting the System under Heavy Pressure. Finally, we run an additional set of experiments that evaluate the impact of metadata distribution on the total aggregated bandwidth under heavy write concurrency, when pushing the pressure on the whole system even further by significantly increasing the number of workers that write job outputs concurrently.

In this setting, we use a larger configuration: we deploy a version manager, a provider manager, 90 data providers and 90 metadata providers. The version manager and the provider manager are deployed on separate physical nodes. The data providers and metadata providers are

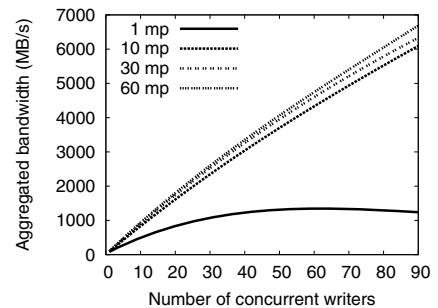


Fig. 5. Impact of the decentralized metadata management on the aggregated bandwidth

co-deployed as in the previous experiment (one provider and one metadata provider per physical node). Finally, a variable number of additional, separate physical nodes (from 1 to 90) host 4 workers each, thus adding up a maximum of 360 workers in the largest configuration. All workers are synchronized to start writing at the same time. Each worker iteratively writes an 8 MB output consisting of 128 pages of 64 KB size. We measure the average write bandwidth per worker over 50 iterations, for all workers and then compute the overall aggregated bandwidth for the whole set of distributed workers.

The results are synthesized on Figure 5. First, we can clearly see that using a centralized metadata provider severely limits the overall aggregated write bandwidth under heavy concurrency, thus demonstrating the benefits of our decentralized metadata management scheme. We could thus measure up to a 6.7 GB/s aggregated bandwidth in a 60-metadata provider configuration, with 360 concurrent writers. Based on the correlation established in the previous experiment, we could estimate that a 7.4 GB/s bandwidth could thus be reached in a scenario where the metadata providers and data providers would be deployed on separate physical nodes. We can further notice the aggregated bandwidth always increases when adding metadata providers, however the improvement is not uniform. Beyond 10 metadata providers (in this experiment), the corresponding performance gain becomes less significant.

6 Conclusion

This paper addresses the problem of data management in Desktop Grids. While classic approaches rely on centralized mechanisms for data management, some recent proposals aim at making Desktop Grids suitable for applications which need to access large amounts of input data. These approaches rely on data distribution using P2P overlays or Content Distribution Networks. We make a step further and propose an approach enabling Desktop Grids to also cope with *write-intensive distributed applications, under potentially heavy concurrency*. Our solution relies on BlobSeer, a blob management service which specifically addresses the issues mentioned above. It implements an efficient storage solution by gathering a *large aggregated storage capacity* from the physical nodes participating to the Desktop Grid.

By combining data fragmentation and striping with an efficient *distributed* metadata management scheme, BlobSeer allows applications to efficiently access data within huge data blobs. The algorithms used by BlobSeer enable a *high write throughput under heavy concurrency*: for any blob update, the new data may asynchronously be sent and stored in parallel on data providers, with no synchronization. Metadata is then also built and stored in parallel, with minimal synchronization. Moreover, BlobSeer provides efficient versioning support thanks to its lock-free design, allowing multiple concurrent writers to efficiently proceed in parallel. Storage is handled in a *space-efficient* way by sharing data and metadata across successive versions. Finally, note that in BlobSeer, accessing data sequentially or randomly has the same cost.

The main contribution of this paper is to explain how the BlobSeer approach fits the needs write-intensive applications and to support our claims through extensive experimental evaluations, which clearly demonstrate the efficiency of our decentralized approach. As a next step, we are currently experimenting efficient ways of using

replication and dynamic group management algorithms, in order to address volatility and fault tolerance issues, equally important in Desktop Grids.

Acknowledgments

This work was supported by the French National Research Agency (LEGO project, ANR-05-CIGC-11). Experiments were carried out using the Grid'5000 experimental testbed, an initiative from the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners (see <http://www.grid5000.fr/>).

References

1. Anderson, D.P.: Boinc: A system for public-resource computing and storage. In: GRID 2004: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing,, Washington, DC, USA, pp. 4–10. IEEE Computer Society, Los Alamitos (2004)
2. Fedak, G., Germain, C., Neri, V.: Xtremweb: A generic global computing system. In: CCGRID 2001: Proceedings of the IEEE International Symposium on Cluster Computing and the Grid. pp. 582–587. Press (2001)
3. Chien, A., Calder, B., Elbert, S., Bhatia, K.: Entropia: Architecture and performance of an enterprise desktop grid system. *Journal of Parallel and Distributed Computing* 63, 597–610 (2003)
4. Costa, F., Silva, L., Fedak, G., Kelley, I.: Optimizing data distribution in desktop grid platforms. *Parallel Processing Letters (PPL)* 18, 391–410 (2008)
5. Al-Kiswany, S., Ripeanu, M., Vazhkudai, S.S., Gharaibeh, A.: Stdchk: A checkpoint storage system for desktop grid computing. In: ICDCS 2008: Proceedings of the the 28th International Conference on Distributed Computing Systems, Washington, DC, USA, pp. 613–624. IEEE Computer Society, Los Alamitos (2008)
6. Lustre file system: High-performance storage architecture and scalable cluster file system. White Paper (2007), http://wiki.lustre.org/index.php/Lustre_Publications
7. Carns, P.H., Ligon III, W.B., Ross, R.B., Thakur, R.: Pvfs: A parallel file system for linux clusters. In: ALS 2000: Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta, GA, USA, pp. 317–327. USENIX Association (2000)
8. Ghemawat, S., Gobioff, H., Leung, S.T.: The google file system. *SIGOPS Oper. Syst. Rev.* 37(5), 29–43 (2003)
9. You, L.L., Pollack, K.T., Long, D.D.E.: Deep store: An archival storage system architecture. In: ICDE 2005: Proceedings of the 21st International Conference on Data Engineering, Washington, DC, USA, pp. 804–8015. IEEE Computer Society, Los Alamitos (2005)
10. Respower render farm (2003), <http://www.respower.com>
11. Patoli, Z., Gkion, M., Al-Barakati, A., Zhang, W., Newbury, P.F., White, M.: How to build an open source render farm based on desktop grid computing. In: Hussain, D.M.A., Rajput, A.Q.K., Chowdhry, B.S., Gee, Q. (eds.) IMTIC. Communications in Computer and Information Science, vol. 20, pp. 268–278. Springer, Heidelberg (2008)
12. Nicolae, B., Antoniu, G., Bougé, L.: How to enable efficient versioning for large object storage under heavy access concurrency. In: EDBT 2009: 2nd International Workshop on Data Management in P2P Systems (DaMap 2009), St Petersburg, Russia (2009)
13. Nicolae, B., Antoniu, G., Bougé, L.: Enabling lock-free concurrent fine-grain access to massive distributed data: Application to supernovae detection. In: CLUSTER 2008: Proceedings of the 2008 IEEE International Conference on Cluster Computing, Tsukuba, Japan, pp. 310–315 (2008)

14. Nicolae, B., Antoniu, G., Boug , L.: Distributed management of massive data: An efficient fine-grain data access scheme. In: Palma, J.M.L.M., Amestoy, P.R., Dayd , M., Mattoso, M., Lopes, J.C. (eds.) VECPAR 2008. LNCS, vol. 5336, pp. 532 543. Springer, Heidelberg (2008)
15. Zheng, C., Shen, G., Li, S., Shenker, S.: Distributed segment tree: Support of range query and cover query over dht. In: IPTPS 2006: The Fifth International Workshop on Peer-to-Peer Systems, Santa Barbara, USA (2006)
16. Boost c++ libraries (2008), <http://www.boost.org>
17. Bolze, R., Cappello, F., Caron, E., Dayd , M., Desprez, F., Jeannot, E., J gou, Y., Lanteri, S., Leduc, J., Melab, N., Mornet, G., Namyst, R., Primet, P., Quetier, B., Richard, O., Talbi, E.G., Touche, I.: Grid'5000: A large scale and highly reconfigurable experimental grid testbed. Int. J. High Perform. Comput. Appl. 20(4), 481 494 (2006)

MapReduce Programming Model for .NET-Based Cloud Computing

Chao Jin and Rajkumar Buyya

Grid Computing and Distributed Systems (GRIDS) Laboratory
Department of Computer Science and Software Engineering
The University of Melbourne, Australia
`{chaojin, raj}@csse.unimelb.edu.au`

Abstract. Recently many large scale computer systems are built in order to meet the high storage and processing demands of compute and data-intensive applications. MapReduce is one of the most popular programming models designed to support the development of such applications. It was initially created by Google for simplifying the development of large scale web search applications in data centers and has been proposed to form the basis of a ‘Data center computer’ This paper presents a realization of MapReduce for .NET-based data centers, including the programming model and the runtime system. The design and implementation of MapReduce.NET are described and its performance evaluation is presented.

1 Introduction

Recently several organizations are building large scale computer systems to meet the increasing demands of high storage and processing requirements of compute and data-intensive applications. On the industry front, companies such as Google and its competitors have constructed large scale data centers to provide stable web search services with fast response and high availability. On the academia front, many scientific research projects increasingly rely on large scale data sets and powerful processing ability provided by super computer systems, commonly referred to as e-Science [15].

These huge demands on data centers motivate the concept of Cloud Computing [9] [12]. With clouds, IT-related capabilities can be provided as service, which is accessible through the Internet. Representative systems include Google App Engine, Amazon Elastic Compute Cloud (EC2), Majrasoft Aneka, and Microsoft Azure. The infrastructure of Cloud Computing can automatically scale up to meet the requests of users. The scalable deployment of applications is typically facilitated by Virtual Machine (VM) technology.

With the increasing popularity of data centers, it is a challenge to provide a proper programming model which is able to support convenient access to the large scale data for performing computations while hiding all low-level details of physical environments. Within all the candidates, MapReduce is one of the most popular programming models designed for this purpose. It was originally proposed by Google to handle large-scale web search applications [8] and has been proved to be an effective programming model for developing data mining and machine learning applications in data centers.

Especially, it can improve the productivity of junior developers who do not have required experiences of distributed/parallel development. Therefore, it has been proposed to form the basis of a ‘data center computer’ [5].

The .NET framework is the standard platform of Microsoft Windows applications and it has been extended to support parallel computing applications. For example, the parallel extension of .NET 4.0 supports the Task Parallel Library and Parallel LINQ, while MPI.NET [6] implements a high performance library for the Message Passing Interface (MPI). Moreover, the Azure cloud service recently released by Microsoft, enables developers to create applications running in the cloud by using the .NET Framework.

This paper presents a realization of MapReduce for the .NET platform, called MapReduce.NET. It not only supports data-intensive applications, but also facilitates a much wider variety of applications, even including some compute-intensive applications, such as Genetic Algorithm (GA) applications. In this paper, we describe:

- MapReduce.NET: A MapReduce programming model designed for the .NET platform using the C# programming language.
- A runtime system of MapReduce.NET deployed in an Enterprise Cloud environment, called Aneka [12].

The remainder of this paper is organized as follows. Section 2 gives an overview of MapReduce. Section 3 discusses related work. Section 4 presents the architecture of MapReduce.NET, while Section 5 discusses the scheduling framework. Section 6 describes the performance evaluation of the system. Section 7 concludes.

2 MapReduce Overview

MapReduce is triggered by the *map* and *reduce* operations in functional languages, such as Lisp. This model abstracts computation problems through two functions: map and reduce. All problems formulated in this way can be parallelized automatically.

Essentially, the MapReduce model allows users to write map/reduce components with functional-style code. These components are then composed as a dataflow graph to explicitly specify their parallelism. Finally, the MapReduce runtime system schedules these components to distributed resources for execution while handling many tough problems: parallelization, network communication, and fault tolerance.

A map function takes a key/value pair as input and produces a list of key/value pairs as output. The type of output key and value can be different from input:

$$map :: (key_1, value_1) \Rightarrow list(key_2, value_2) \quad (1)$$

A reduce function takes a key and associated value list as input and generates a list of new values as output:

$$reduce :: (key_2, list(value_2)) \Rightarrow list(value_3) \quad (2)$$

A MapReduce application is executed in a parallel manner through two phases. In the first phase, all map operations can be executed independently from each other. In the second phase, each reduce operation may depend on the outputs generated by any number of map operations. All reduce operations can also be executed independently similar to map operations.

3 Related Work

Since MapReduce was proposed by Google as a programming model for developing distributed data intensive applications in data centers, it has received much attention from the computing industry and academia. Many projects are exploring ways to support MapReduce on various types of distributed architecture and for a wider range of applications. For instance, Hadoop [2] is an open source implementation of MapReduce sponsored by Yahoo!. Phoenix [4] implemented the MapReduce model for the shared memory architecture, while M. Kruijf and K. Sankaralingam implemented MapReduce for the Cell B.E. architecture [11].

A team from Yahoo! research group made an extension on MapReduce by adding a merge phase after reduce, called Map-Reduce-Merge [7], to perform join operations for multiple related datasets. Dryad [10] supports an interface to compose a Directed Acyclic Graph (DAG) for data parallel applications, which can facilitate much more complex components than MapReduce.

Other efforts focus on enabling MapReduce to support a wider range of applications. MRPSO [1] utilizes the Hadoop implementation of MapReduce to parallelize a compute-intensive application, called Particle Swarm Optimization. Researchers from Intel currently work on making MapReduce suitable for performing earthquake simulation, image processing and general machine learning computations [14]. MRPGA [3] is an extension of MapReduce for GA applications based on MapReduce.NET. Data-Intensive Scalable Computing (DISC) [13] started to explore suitable programming models for data-intensive computations by using MapReduce.

4 Architecture

MapReduce.NET resembles Google's MapReduce, but with special emphasis on the .NET and Windows platform. The design of MapReduce.NET aims to reuse as many existing Windows components as possible. Fig. 1 illustrates the architecture of MapReduce.NET. Its implementation is assisted by several component services from Aneka. Aneka is a .NET-based platform for enterprise and public Cloud Computing [12]. It supports the development and deployment of .NET-based Cloud applications in public Cloud environments, such as Amazon EC2. We used Aneka to simplify the deployment of MapReduce.NET in distributed environments. Each Aneka node consists of a configurable container, hosting mandatory and optional services. The mandatory services provide the basic capabilities required in a distributed system, such as communications between Aneka nodes, security, and membership. Optional services can be installed to support the implementation of different programming models in Cloud environments. MapReduce.NET is implemented as an optional service of Aneka.

Besides Aneka, WinDFS provides a distributed storage service over the .NET platform. WinDFS organizes the disk spaces on all the available resources as a virtual storage pool and provides an object-based interface with a flat name space, which is used to manage data stored in it. To process local files, MapReduce.NET can also directly communicate with CIFS or NTFS. The remainder of this section presents details on the programming model and runtime system.

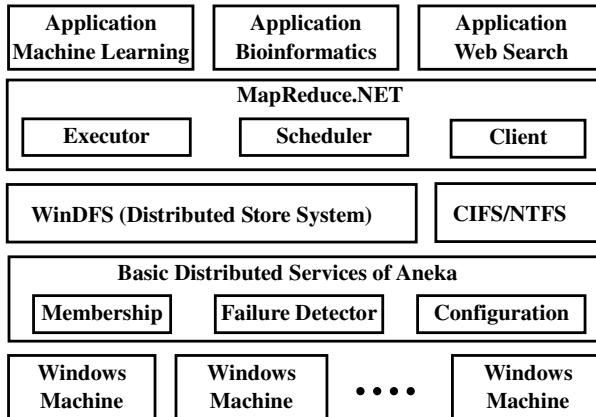


Fig. 1. Architecture of MapReduce.NET

Table 1. APIs of MapReduce.NET

```

class Mapper
{
    void Map(MapInput < K, V > input)
}

class Reducer
{
    void Reduce(IReduceEnumerator input)
}

```

4.1 MapReduce.NET APIs

The implementation of MapReduce.NET exposes APIs similar to Google MapReduce. Table 1 illustrates the interface presented to users in the C# language. To define map/reduce functions, users need to inherit from *Mapper* or *Reducer* class and override corresponding abstract functions. To execute the MapReduce application, the user first needs to create a *MapReduceApp* class and set it with the corresponding *Mapper* and *Reducer* classes. Then, input files should be configured before starting the execution and they can be local files or files in the distributed store.

The type of input key and value to the *Map* function is the *object*, which is the root type of all types in C#. For reduce function, the input is organized as a collection and the data type is *IEnumerable*, which is an interface for supporting an iterative operation on the collection. The data type of each value in the collection is also *object*.

With *object*, any type of data, including user-defined or system build-in type, can be accepted as input. However, for user defined types, users need to provide serialization and deserialization methods. Otherwise, the default serialization and deserialization methods will be invoked.

4.2 Runtime System

The execution of a MapReduce.NET application consists of 4 major phases: Map, Sort, Merge and Reduce. The overall flow of execution is illustrated in Fig. 2. The execution starts with the Map phase. It iterates the input key/value pairs and invokes the map function defined by users on each pair. The generated results are passed to the Sort and Merge phases, which perform sorting and merging operations to group the values with identical keys. The result is an array, each element of which is a group of values for each key. Finally, the Reduce phase takes the array as input and invokes the reduce function defined by users on each element of the array.

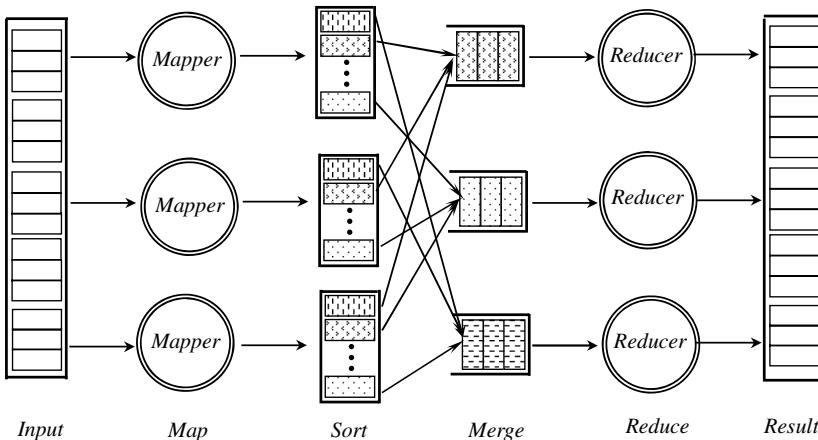


Fig. 2. Computation of MapReduce.NET

The runtime system is based on the master-slave architecture with the execution of MapReduce.NET orchestrated by a scheduler. The scheduler is implemented as a MapReduce.NET Scheduler service in Aneka, while all the 4 major phases are implemented as a MapReduce.NET Executor service. With Aneka, the MapReduce.NET system can be deployed in cluster or data center environments. Typically, it consists of one master machine for a scheduler service and multiple worker machines for executor services.

The 4 major phases are grouped into two tasks: Map task and Reduce task. The Map task executes the first 2 phases: map and sort, while the Reduce task executes the last 2 phases: merge and reduce. The input data for the map function is split into even-sized m pieces to be processed by m map tasks, which are evenly assigned to worker computers. The intermediate results generated by map tasks are partitioned into r fragments, and each fragment is processed by one reduce task.

The major phases on the MapReduce.NET executor are illustrated in Fig. 3.

Map Phase. The executor extracts each input key/value pair from the input file. For each key/value pair, it invokes the map function defined by users. The result generated by the map function is first buffered in the memory. The memory buffer consists of

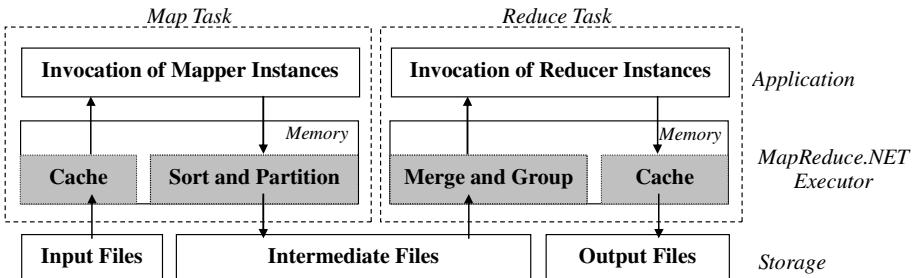


Fig. 3. Dataflow of MapReduce.NET

many buckets and each one is for a different partition. The generated result determines its partition through a hash function, which may be defined by users. Then the result is appended to the tail of the bucket of its partition. When the size of all the results buffered in the memory reaches a predefined maximal threshold, they are sent to the Sort phase and then written to the disk. This saves space for holding intermediate results for the next round of map invocations.

Sort Phase. When the size of buffered results exceeds the maximal threshold, each bucket is written to disk as an intermediate file. Before the buffered results are written to disk, elements in each bucket are sorted in memory. They are written to disk by the sorted order, either ascending or descending. The sorting algorithm adopted is quick sort.

Merge Phase. To prepare inputs for the Reduce phase, we need to merge all the intermediate files for each partition. First, the executor fetches intermediate files which are generated in the Map phase from neighboring machines. Then, they are merged to group values with the same key. Since all the key/value pairs in the intermediate files are already in a sorted order, we deploy a heap sort to achieve the group operation. Each node in the heap corresponds to one intermediate file. Repeatedly, the key/value pair on the top node is picked, and simultaneously the values associated with same key are grouped.

Reduce Phase. In our implementation, the Reduce phase is combined with the Merge phase. During the process of heap sort, we combine all the values associated with the same key and then invoke the reduce function defined by users to perform the reduction operation on these values. All the results generated by reduce function are written to disk according to the order by which they are generated.

4.3 Memory Management

Managing memory efficiently is critical for the performance of applications. On each executor, the memory consumed by MapReduce.NET mainly includes memory buffers for intermediate results, memory space for the sorting algorithm and buffers for input and output files. The memory management is illustrated in Fig. 3.

The system administrator can specify a maximal value for the size of memory used by MapReduce.NET. This size is normally determined by the physical configuration of machines and the memory requirement of applications.

According to this maximal memory configuration, we set the memory buffer used by intermediate results and input/output files. The default value for read/write buffer of each file is 16MB. The input and output files are from the local disk. Therefore, we use the *FileStream* class to control the access to local files.

The memory buffer for intermediate results is implemented by using the *MemoryStream* class, which is a stream in memory. All the results generated by map functions are serialized and then append to the tail of the stream in memory.

5 Scheduling Framework

This section describes the scheduling model for coordinating multiple resources to execute MapReduce computations. The scheduling is managed by the MapReduce.NET scheduler. After users submit MapReduce.NET applications to the scheduler, it maps Map and Reduce tasks to different resources. During the execution, it monitors the progress of each task and migrate tasks when some nodes are much slower than others due to their heterogeneity or interference of dominating users.

Typically, a MapReduce.NET job consists of m Map tasks and r Reduce tasks. Each Map task has an input file and generates r result files. Each Reduce task has m input files which are generated by m Map tasks.

Normally the input files for Map tasks are available in WinDFS or CIFS prior to job execution, thus the size of each Map input file can be determined before scheduling. However, the output files are dynamically generated by Map tasks during execution, hence the size of these output files is difficult to determine prior to job execution.

The system aims to be deployed in an Enterprise Cloud environment, which essentially organizes idle resources within a company or department as a virtual super computer. Normally, resources in Enterprise Clouds are shared by the owner of resources and the users of idle resources. The latter one should not disturb the normal usage of resource owner. Therefore, with an Enterprise Cloud, besides facing the traditional problems of distributed system, such as complex communications and failures, we have to face *soft failure*. Soft failure refers to a resource involved in MapReduce execution having to quit computation due to domination by its owner.

Due to the above dynamic features of MapReduce.NET application and Enterprise Cloud environments, we did not choose a static scheduling algorithm. On the contrary, the basic scheduling framework works like the work-stealing model. Whenever a worker node is idle, a new Map or Reduce task is assigned to it for execution with special priority on taking advantage of data locality.

The scheduling algorithm starts with dispatching Map tasks as independent tasks. The Reduce tasks, however, are dependent on the Map tasks. Whenever a Reduce task is ready (i.e. all its inputs are generated by Map tasks), it will be scheduled according to the status of resources. The scheduling algorithm aims to optimize the execution time, which is achieved by minimizing Map and Reduce tasks respectively.

6 Performance Evaluation

We have implemented the programming model and runtime system of MapReduce.NET and deployed it on desktop machines of several student laboratories in Melbourne University. This section evaluates its performance based on two benchmark applications: Word Count (WC) and Distributed Sort (DS).

All the experiments were executed in an Enterprise Cloud consisting of 33 machines located in 3 student laboratories. For distributed experiments, one machine was set as master and the rest were configured as worker machines. Each machine has a single Pentium 4 processor, 1GMB memory, 160GB hard disk (10GB is dedicated for WinDFS storage), 1 Gbps Ethernet network and runs Windows XP.

6.1 Sample Applications

The sample applications (WC and DS) are benchmarks used by Google MapReduce and Hadoop. To implement the WC application, users just need to split words for each text file in the map function and sum the number of appearance for each word in the reduce function. For the DS application, users do not have to do anything within the map and reduce functions, while MapReduce.NET performs sorting automatically.

The rest of this section presents the overhead of MapReduce.NET. First, we show the overhead caused by the MapReduce programming model in a local execution. Then the overhead of MapReduce.NET in a distributed environment is reported.

6.2 System Overhead

MapReduce can be regarded as a parallel design pattern, which trades performance to improve the simplicity of programming. Essentially, the Sort and Merge phases of the MapReduce runtime system introduce extra overhead. However, the sacrificed performance cannot be overwhelming. Otherwise, it would not be acceptable. We evaluate the overhead of MapReduce.NET with local execution. The input files are located on the local disk and all 4 major phases of MapReduce.NET executes sequentially on a single machine. This is called a local runner and can be used for debugging purposes.

For local execution, both sample applications were configured as follows:

- The WC application processes the example text files used by Phoenix [1] and the size of raw data 1GB.
- The DS application sorts a number of records consisting of a key and a value, both of which are random integers. The input data includes 1,000 million records with 1.48GB raw data.

The execution time is split into 3 parts: Map, Sort and Merge+Reduce. They correspond to the time consumed by reading inputs and invoking map functions, the time consumed by the sort phase(including writing intermediate results to disk) and the time consumed by the Reduce tasks. In this section, we analyze the impact of buffer size for intermediate results on the execution time of applications. In particular, the experiments were executed with different sizes of memory buffer for intermediate results. The size

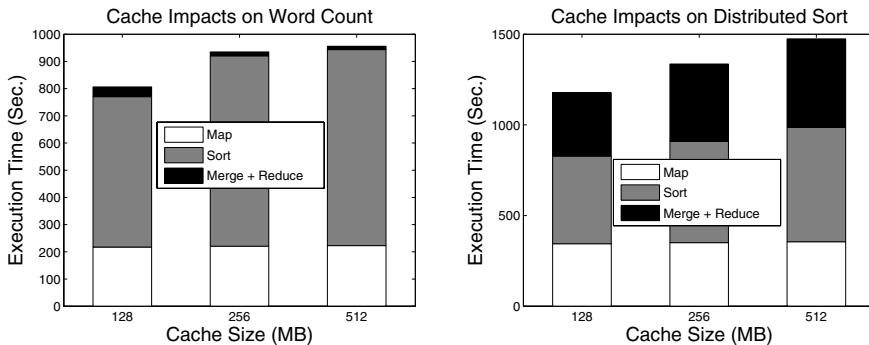


Fig. 4. Cache Impacts of MapReduce.NET

of memory buffer containing intermediate results was set to be 128MB, 256MB and 512MB respectively and the results for both applications are shown in Fig. 4.

First, we can see that different types of application have different percentage distribution for each part. For the WC application, the time consumed by the reduce and merge phases can even be ignored. The reason is that the size of results of WC is comparatively small. On the contrary, the reduce and merge phases of the DS application incur a much larger percentage of total time consumed.

Second, out of our expectation, increasing the size of the buffer for intermediate results may not reduce the execution time for both applications. On the contrary, a larger buffer increases the time consumed by sorting because sorting more intermediates at one time needs deeper stack and more resources. A larger memory buffer generates fewer intermediate files, but each is characterized by a larger size. The read/write buffer of each input/output files is configured per file, and the default value is 16MB. Therefore, with a larger buffer for intermediate results in the Map and Sort phase, the Reduce phase consumes longer time because the overall size of of input file buffers is smaller. However, a larger memory buffer does not have significant impacts on the Map phase.

6.3 Overhead Comparison with Hadoop

This section compares the overhead of MapReduce.NET with Hadoop, an open source implementation of MapReduce in Java. Hadoop is supported by Yahoo! and aims to be a general purpose distributed platform. We use the latest stable release of Hadoop (version 0.18.3).

To compare the overhead, we run the local runner of Hadoop and MapReduce.NET respectively with the same input size for both applications. The size of buffer for intermediate results was configured to be 128MB for both implementations. The configuration of WC and DS applications are the same as Section 6.2. The JVM adopted in our experiment is Sun JRE1.6.0, while the version of the .NET framework is 2.0. The results are shown in Fig. 5. MapReduce.NET performs better than Hadoop for both applications. Specifically, both *Map* and *Merge+Reduce* phase of MapReduce.NET consumes less time than Hadoop, but more time than Hadoop in the *Sort* phase.

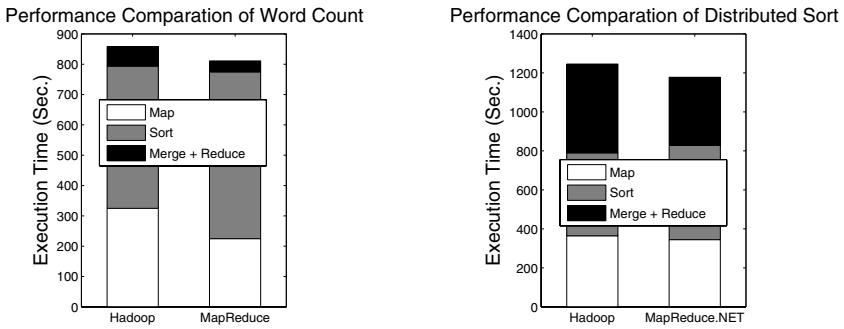


Fig. 5. Overhead Comparison of Hadoop and MapReduce.NET

Reasons for this are: (a) the deserialization and serialization operations achieved by MapReduce.NET is more efficient than Hadoop; (b) the Merge phase of Hadoop involves extra IO operations than MapReduce.NET. In particular, for the Map phase, the major overhead of both applications consists of invocation of deserialization of raw input data and map functions combined with reading disk operations. According to our experiments, however, we did not find significant performance difference of disk IO operations by using JRE1.6.0 and .NET 2.0 over Windows XP. In the Merge+Reduce phase, the major overhead includes serialization, deserialization and reading and writing disk. Hadoop splits the large input files into a number of small pieces (32 pieces for WC and 49 pieces for DS) and each piece corresponds to a Map task. Then, Hadoop first has to merge all the intermediate results for the same partition from multiple Map tasks prior to starting the combined Merge and Reduce phase. MapReduce.NET does not require this extra overhead. Therefore it performs better than Hadoop in the *Merge+Reduce* phase.

In the Sort phase, the sorting algorithm implemented by Hadoop is more efficient than its corresponding implementation in MapReduce.NET. Both MapReduce.NET and Hadoop implement the same sorting algorithm, hence identifying the difference in performance between two implementations implies a deep investigation involving the internals of the two virtual machines.

6.4 System Scalability

In this section, we evaluate the scalable performance of MapReduce.NET in a distributed environment. Applications were configured as follows:

- WC: We duplicated the original text files used by Phoenix [1] to generate an example input with 6GB raw data, which is split into 32 files.
- DS: sorts 5,000 million records in an ascending order. The key of each record is a random integer. The total raw data is about 7.6GB, which is partitioned into 32 files.

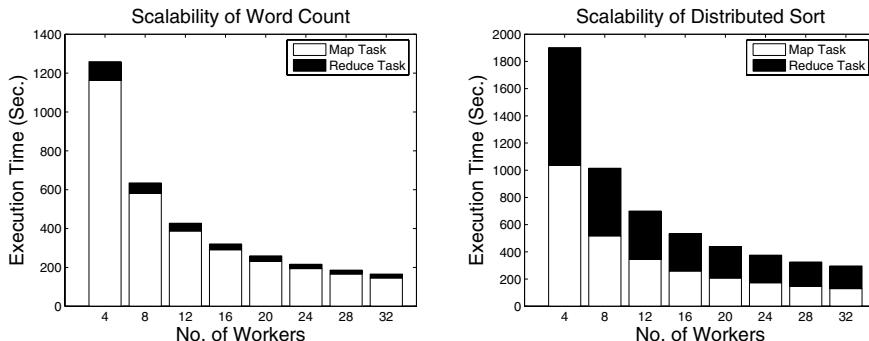


Fig. 6. Scalability of MapReduce.NET in Aneka Environment

Fig. 6 illustrates the scalable performance result of the WC application. The Map task execution time consists of the map and sort phases which include the time from starting the execution to the end of execution for all Map tasks. The Reduce task execution time consists of overhead of network traffic, the merge phase and the reduce phase invoking reduce functions on all the worker machines. We can see that the map and sort phases dominate the entire execution of the WC application.

Unlike the WC application, the DS application has a nearly uniform distribution of execution time for Map and Reduce tasks, as shown in Fig. 6. The network traffic also incurs a substantial percentage of the entire execution, because the intermediate results are actually the same as the original input data.

For both applications, the performance increases as more resources are added to the computation. Therefore, MapReduce.NET is able to provide scalable performance within homogenous environments when the number of machines increases.

7 Conclusions

This paper presented MapReduce.NET, a realization of MapReduce on the .NET platform. It provides a convenient interface to access large scale data in .NET-based distributed environments. We evaluated the overhead of our implementation on Windows platforms. Experimental results have shown that the performance of MapReduce.NET is comparable or even better (for some cases) than Hadoop on WindowsXP. Furthermore, MapReduce.NET provides scalable performance in distributed environments. Hence, MapReduce.NET is practical for usage as a general purpose .NET-based distributed and Cloud computing model. In the future, we endeavor to integrate MapReduce.NET with the Azure Cloud Platform.

Acknowledgements

This work is partially supported by research grants from the Australian Research Council (ARC) and Australian Department of Industry, Innovation, Science and Research (DIISR). We would like to thank Chee Shin Yeo, Christian Vecchiola and Jemal Abawajy for their comments on improving the quality of the paper.

References

- [1] McNabb, A.W., Monson, C.K., Seppi, K.D.: Parallel PSO Using MapReduce. In: Proc. of the Congress on Evolutionary Computation (2007)
- [2] Apache Hadoop, <http://lucene.apache.org/hadoop/>
- [3] Jin, C., Vecchiola, C., Buyya, R.: MRPGA: An Extension of MapReduce for Parallelizing Genetic Algorithms. In: Proc. of 4th International Conference on e-Science (2008)
- [4] Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., Kozyrakis, C.: Evaluating MapReduce for Multi-core and Multiprocessor Systems. In: Proc. of the 13th Intl. Symposium on High-Performance Computer Architecture (2007)
- [5] Patterson, D.A.: Technical perspective: the data center is the computer. Communications of the ACM 51(1), 105 (2008)
- [6] Gregor, D., Lumsdaine, A.: Design and Implementation of a High-Performance MPI for C# and the Common Language Infrastructure. In: Proc. of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (2008)
- [7] Yang, H.C., Dasdan, A., Hsiao, R.L., Stott Parker, D.: Map-Reduce-Merge: simplified relational data processing on large clusters. In: Proc. of SIGMOD (2007)
- [8] Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: Proc. of the 6th Symposium on Operating System Design and Implementation (2004)
- [9] Varia, J.: Cloud Architectures. White Paper of Amazon (2008),
<jineshvaria.s3.amazonaws.com/public/>
<cloudarchitectures-varia.pdf>
- [10] Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In: Proc. of European Conference on Computer Systems, EuroSys (2007)
- [11] Kruijff, M., Sankaralingam, K.: MapReduce for the Cell B.E. Architecture. TR1625, Technical Report, The University of Wisconsin-Madison (2007)
- [12] Buyya, R., Yeo, C.S., Venugopal, S., Broberg, J., Brandic, I.: Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility. Future Generation Computer Systems 25(6), 599–616 (2009)
- [13] Bryant, R.E.: Data-Intensive Supercomputing: The Case for DISC. CMU-CS-07-128, Technical Report, Carnegie Mellon University (2007)
- [14] Chen, S., Schlosser, S.W.: Map-Reduce Meets Wider Varieties of Applications. IRP-TR-08-05, Technical Report, Intel Research Pittsburgh (2008)
- [15] Hey, T., Trefethen, A.: The data deluge: an e-Science perspective. In: Grid Computing: Making the Global Infrastructure a Reality, pp. 809–824 (2003)

The Architecture of the XtreemOS Grid Checkpointing Service

John Mehnert-Spahn¹, Thomas Ropars², Michael Schoettner¹,
and Christine Morin³

¹ Department of Computer Science, Heinrich-Heine University, Duesseldorf, Germany
{John.Mehnert-Spahn,Michael.Schoettner}@uni-duesseldorf.de

² Université de Rennes 1, IRISA, Rennes, France
Thomas.Ropars@irisa.fr

³ INRIA Centre Rennes - Bretagne Atlantique, Rennes, France
Christine.Morin@inria.fr

Abstract. The EU-funded XtreemOS project implements a grid operating system (OS) transparently exploiting distributed resources through the SAGA and POSIX interfaces. XtreemOS uses an integrated grid checkpointing service (XtreemGCP) for implementing migration and fault tolerance for grid applications. Checkpointing and restarting applications in a grid requires saving and restoring distributed/parallel applications in distributed heterogeneous environments. In this paper we present the architecture of the XtreemGCP service integrating existing system-specific checkpointer solutions. We propose to bridge the gap between grid semantics and system-specific checkpointers by introducing a common kernel checkpointer API that allows using different checkpointers in a uniform way. Our architecture is open to support different checkpointing strategies that can be adapted according to evolving failure situations or changing application requirements. Finally, we discuss measurements numbers showing that the XtreemGGP architecture introduces only minimal overhead.

1 Introduction

Grid technologies like Globus [5] can help to run businesses in dynamic distributed environments effectively. Grids enable/simplify the secure and efficient sharing and aggregation of resources across administrative domains spawning millions of nodes and thousands of users.

The key idea of the EU-funded XtreemOS project is to reduce the burden on grid administrators and application developers by providing a Linux-based open source Grid operating system (OS). XtreemOS provides a sound base for simplifying the implementation of higher-level Grid services because they can rely on important native distributed OS services, e.g. security, resource, and process management [1]. In XtreemOS a grid node may be a single PC, a LinuxSSI cluster, or a mobile device. LinuxSSI is an extension of the Linux-based Ker-righed single system image operating system [17]. Because of resource constraints mobile nodes act as clients rather than fully fledged grid nodes.

Grid technologies offer a great variety of benefits but there are still challenges ahead including fault tolerance. With the increasing number of nodes more resources become available but at the same time the failure probability increases. Fault tolerance can be achieved for many applications, in particular scientific applications, using a rollback-recovery strategy. In the simplest scenario an application is halted and a checkpoint is taken that is sufficient to restart the application in the event of a failure. Beyond using checkpointing for fault tolerance it is also a basic building block for application migration, e.g. used for load balancing.

There are numerous state of the art system-specific checkpointing solutions supporting checkpointing and restart on single Linux machines, e.g. BCLR [8], Condor [10], libCkpt [13]. They have different capabilities regarding what kind of resources can be checkpointed. Furthermore, there are also specific distributed checkpointing solutions for cluster systems dealing with communication channels and cluster-wide shared resources, like the Kerrighed checkpointer [17].

The contribution of this paper is to propose an architecture for the Xtreem-GCP service integrating different existing checkpointing solutions in order to checkpoint grid applications. This is realized by introducing a common kernel checkpointer API that is implemented by customized translation libraries. The architecture has been implemented currently supporting the BLCR and LinuxSSI checkpointer within the XtreemOS project which is available as open source [1].

The outline of this paper is as follows. In Section 2 we briefly present relevant background information on XtreemOS. Subsequently, we present the architecture of XtreemGCP. In Section 4 we discuss the common kernel checkpointer API and implementation aspects. Other grid related issues including resource conflicts and security are described in Section 5 followed by an evaluation. Related work is discussed in Section 7 followed by conclusions and future work.

2 Checkpointing an Application in XtreemOS

In this paper, we consider sequential, parallel, and distributed applications. We call a job an application that has been submitted to the grid. Each job has a grid-wide unique job id. A job is divided into job-units, a job-unit being the set of processes of a job running on one grid node.

Applications are handled by the application execution management service (AEM). AEM job managers handle jobs life cycle, including submission, execution and termination. Each job is managed by one job manager. A job directory service stores the list of active jobs and the location of their associated job managers. On each grid node, an execution manager handles the job-units running on the node. It performs the actions requested by the job manager and is in charge of controlling and managing job-units.

XtreemOS includes a grid file system called XtreemFS [6], providing location-transparent file access (including replication and striping). XtreemGCP uses XtreemFS as persistent storage for grid checkpoint files.

Checkpoint/restart is of major importance in XtreemOS - it is the basic block for several functionalities: *scheduling*: including job migration and job

suspension, *fault tolerance*: implemented by saving a job snapshot, and *debugging*: allowing to run applications in the past.

3 Architecture of XtreemGCP

The XtreemGCP service architecture, its components and their interactions are explained here in the context of checkpointing and restarting grid applications.

3.1 Grid Checkpointing Services

The XtreemGCP is a layered architecture, see Figure 1. At the grid level, a job checkpoint (JCP) manages checkpoint/restart for one job possibly distributed over many grid nodes. Therefore, it uses the services provided by the job-unit checkpoint (JUCP) available on each grid node. A JUCP controls the kernel checkpointers available on the grid nodes to take snapshots of the job-units processes. The so-called *common kernel checkpoint API*, introduced by XtreemOS, enables the JUCP to address any underlying kernel checkpoint in a transparent way. A translation library implements this API for a specific kernel checkpoint and translates grid semantics into specific kernel checkpoint semantics. In the following we provide details for all these services.

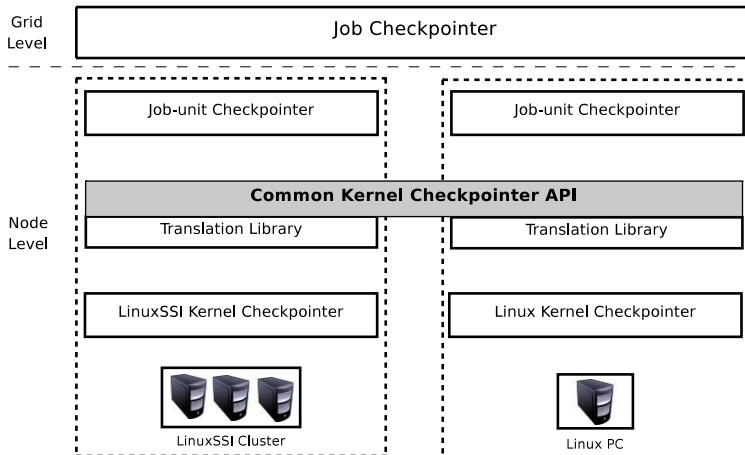


Fig. 1. XtreemOS Grid Checkpointing Service Architecture

There is one **job checkpoint** (JCP) service per job in the grid. It is located on the same node as the corresponding job manager. Furthermore, it distributes the load, induced by XtreemGCP over the whole grid. We use virtual nodes [14] to make JCPs and job managers highly available via service replication.

The JCP has a global view of the job. It knows the job-units composing the job and the job-units location. It controls the underlying components to apply a checkpoint decision. When checkpointing is used to migrate or suspend a job,

the decision is taken by the AEM. In case of checkpoints used for fault tolerance, the JCP is responsible for scheduling the checkpoints.

In coordinated checkpointing, the JCP takes the role of the coordinator. With the help of the JUCPs, it synchronizes the job-units at checkpoint time to guarantee a consistent checkpoint. At restart, it coordinates the job-units to ensure a consistent job state, see Section 3.2.

Because of its global view on the job, the JCP is in charge of creating job checkpoint meta-data. The latter describes the job checkpoint image required at restart to recreate a job and its constituting job-units, see Section 5.2.

There is one **job-unit checkpoint** (JUCP) per grid node. It interacts with the JCP to apply a checkpoint decision. The JUCP is in charge of checkpointing job-units. Therefore, it relies on the common kernel checkpointing API to transparently address a specific kernel checkpointer.

Each grid node comes with heterogenous kernel checkpointers having different calling interfaces, checkpoint capabilities, process grouping techniques, etc. One grid node may also offer several kernel checkpointers. Therefore, we introduce the **common kernel checkpointer API**, see Section 4, that allows the JUCP to uniformly access different kernel checkpointers. A translation library implements the common kernel checkpointer API for one kernel checkpointer. Such a kernel checkpointer-bound translation library is dynamically loaded by the JUCP when the kernel checkpointer is required to checkpoint/restart a job-unit. In general the translation library converts grid semantics into a kernel checkpointer semantics, e.g. a grid user ID is translated into the local user id.

A **kernel checkpoint** is able to take a snapshot of a process group, including process memory and diverse process resources, and is able to restart the process group from this snapshot. Unlike checkpointers exclusively implemented in user space, a kernel checkpoint is also capable of restoring kernel structures representing process' resources like process ids or session ids. XtreemGCP also supports virtual machines to be used as kernel checkpointer.

3.2 Checkpointing Strategies

In this section we present how our architecture supports different checkpointing strategies including coordinated and uncoordinated ones. Here we only focus on the interactions between the services. Other issues related to checkpointing, like security or checkpoint file management, are described in Section 5.

Coordinated checkpointing is the protocol used to realise job migration and suspension. In coordinated checkpointing, the job takes the role of the coordinator. It addresses those JUCPs residing on the same grid nodes as the job-units of the job to be checkpointed. Upon receiving the job *checkpoint signal* the JCP requests all involved JUCPs to load the appropriate translation library whose associated kernel checkpointer is capable of checkpointing the job. Afterwards the JCP enters the *prepare phase*. A job is notified of an upcoming system-initiated checkpoint action to prepare itself, e.g. an interactive application notifies the user to be temporarily unavailable. Then, the JUCPs extract the restart-relevant meta-data, described in Section 5.2, with the help of

the kernel checkpointers storing this meta-data in XtreemFS. Afterwards the JCP instructs the JUCPs to enter the *stop phase*. Processes of all involved job-units are requested to execute their checkpoint-related callbacks, see Section 4.3. Then, each job-unit process is stopped from further execution and an acknowledgement is sent by the JUCPs to the JCP. Meta-data are updated in case processes have been created or destroyed between initial saving and process synchronization. The JCP waits for all JUCP acknowledgments to enter the *checkpoint phase*. The kernel checkpointers are requested through the JUCPs to take snapshots of the processes. These snapshots are saved onto XtreemFS. Afterwards the *resume phase* is entered. At *restart* the *rebuild phase* is entered. The JCP parses the checkpoint meta-data of the job to be restarted. This data is used by the job manager, to identify grid nodes with spare resources (PIDs, shmIDs, etc.) being valid before checkpointing in order to reallocate them, and the JUCPs, to select the same kernel checkpointers as the ones used at checkpoint time. Afterwards, the JCP informs all involved JUCPs of the job-units belonging to the job that is to be restarted. Each JUCP gets the JUCP image it needs from XtreemFS and rebuilds the job-unit processes with the help of the kernel checkpointers. Subsequently, the JUCPs enter the *resume phase*. Here all restart callbacks are processed and afterwards the job-units resume with normal execution.

In **uncoordinated checkpointing**, checkpoints of job-units are taken independently thus avoiding synchronization overhead at checkpoint time. The decision to trigger checkpoints is shifted from the JCP to the JUCPs. They can take node and job-unit state-information into account to decide when it is opportune to checkpoint. The common kernel checkpoint API also allows an application to checkpoint a job-unit.

At restart, a consistent global state of the application has to be computed. Therefore dependencies between checkpoints have to be logged during failure-free execution. As described in Section 4.4, the AEM service monitors the processes to provide information about their communications. The involved JUCPs are in charge of analyzing this data in order to compute the dependencies between checkpoints. At restart, the JCP uses the dependency data saved with the checkpoints to compute a consistent global state. Then, it informs the appropriate JUCPs to perform the restart procedure. JUCPs could log messages to avoid the domino effect.

In **application-level checkpointing**, the application itself executes the checkpoint and recovery functions. Here the checkpointing code is embedded in the application, written by the programmer. In case of application-level checkpointing, the application interacts with the JUCP(s). MPI has become a de-facto standard for high performance computing applications. Several MPI libraries provide fault-tolerance mechanisms. Open MPI [7] and LAM/MPI [15] have chosen to implement generic checkpoint/restart mechanisms that can support multiple existing kernel checkpointers. Since this approach fits well with ours, these libraries can be supported by our service, too. When an MPI library is used, the responsibility to coordinate the processes or to detect dependencies is

shifted to this library. On the grid node where the *mpirun* has been launched the JUCP merely initiates checkpointing and manages the checkpoint images created.

3.3 XtreemOS Adaptive Checkpoint Policy Management

Since the grid is a dynamic environment and applications consume resources in an unpredictable manner, XtreemGCP must be able to dynamically adapt its checkpointing policy for efficiency reasons.

As described before, XtreemGCP supports several checkpointing strategies. Both, coordinated and uncoordinated checkpointing, have pros and cons. Adapting checkpointing parameters or switching from one strategy to another depends on various factors, e.g. failure frequency, job behaviour, etc. strategy changes can improve the performance, see [4].

4 Common Kernel Checkpointer API

Obviously, we cannot assume that each grid application or grid node uses the same kernel checkpointer. The *common kernel checkpointer API* bridges differences allowing the JUCPs to uniformly access these different kernel checkpointers. Currently, we have implemented translation libraries for the BLCR and LinuxSSI checkpointers. The translation libraries are able to address user space components and kernel components, too. The following sections give an overview of the API and discuss important aspects of the translation libraries.

4.1 API Overview

The *common kernel checkpointer API* is implemented by the dynamically loadable translation library per kernel checkpointer. The current version of the implementation of the API provides following primitives:

- *xos_prepare_environment*: A process group is searched, containing all processes of job-unit, that can be addressed by the underlying kernel checkpointer. If successful, the process group reference id and type are saved by the JCP with the checkpoint meta-data.
- *xos_stop_job_unit* The execution of checkpoint callbacks is enforced and afterwards, all processes belonging to the job-unit are synchronized.
- *xos_checkpoint_job_unit* A process group checkpoint is taken and saved in a XtreemFS volume.
- *xos_resume_job_unit_cp* All stopped processes are waked up and resume their execution.
- *xos_rebuild_job_unit* The previously stored checkpoint meta-data indicate the appropriate XtreemFS volumes that stores the JCP image of a given checkpoint version number. The checkpointer is instructed to rebuild all structures representing the process group. Processes are still stopped to avoid any interferences.
- *xos_resume_job_unit_rst* The rebuilded processes are waked up and proceed with normal execution. Execution of restart callbacks is enforced.

4.2 Process Groups

Different process grouping semantics are used at AEM and kernel checkpointer level. Instead of a process list, checkpointers like BLCR use an identifier to reference a UNIX process group (addressed via PGID), a UNIX session (addressed via SID), a root-process of a process tree. LinuxSSI uses a LinuxSSI application identifier to checkpoint/restart processes. However, AEM uses only the concept of job-unit (addressed by a jobID) to group processes. Since the AEM JCP initiates checkpoint/restart, it merely has a jobID. In general, if process groups used by AEM and kernel checkpointers mismatch, too many or just a subset of the processes are checkpointed/restarted which results in inconsistencies. Obviously, AEM cannot be hardcoded against one process group type, since various kernel checkpointers use different process groups semantics.

In XtreemGCP this issue is solved by the translation library which ensures, that a kernel checkpointer can reference all processes that AEM captures in a job-unit. In terms of LinuxSSI the translation library manages to put all job-unit processes into an SSI application. The SSI application ID is then determined and can be used for checkpoint/restart. In terms of BLCR the root process is determined based upon the job-unit process list. Since no separate UNIX session or UNIX process group is initiated at job submission, BLCR cannot use these two process group semantics. More information can be found under [\[11\]](#).

Using the root-process of a process tree is not safe. If an intermediate process fails, the subtree cannot be referenced anymore by the root-process. Linux provides so-called *cgroups* [\[3\]](#) - a mechanism to group tasks and to apply special behaviours onto them. *Cgroups* can be managed from user space via a dedicated file system. Since the upcoming Linux-native checkpointer will be based on *cgroups* and LinuxSSI will be ported towards using *cgroups* - a commonly used process group semantic on user and kernel checkpointer level has been found which avoids the occurrence of mismatchings. *Cgroups* allow all processes to be referenced by one identifier even in case of the application internally establishing multiple UNIX process groups or sessions and in case of intermediate process failures.

Integration of *cgroups* is planned by XtreemOS as soon as the Linux-native checkpointer is available. Management of *cgroups* will also be done within a kernel checkpointer-bound translation library.

4.3 Callbacks

As mentioned before different kernel checkpointers vary in their capability to record and re-establish application states. On the one hand limitations of kernel checkpointers must be handled, e.g. resources that cannot be checkpointed. On the other hand they must be prevented from saving too much to be efficient, e.g. it might not be necessary to save very large files with each checkpoint.

Our solution for both aspects is to provide callback functions for application developers, that are executed before and after checkpointing (pre/post-checkpoint callbacks) and after restart. Resources can be saved in an application-customized

way and unnecessary overhead can be reduced, e.g. large files can be closed during checkpointing and re-opened at restart. Integration of callbacks into the execution of checkpoint/restart has been proposed and implemented by BLCR, [15]. We have extended LinuxSSI towards this functionality, too.

To hide kernel checkpointer related semantics of callback management, the common kernel checkpointer API also provides an interface to applications for de/registering callbacks.

4.4 Communication Channels

In order to take consistent distributed snapshots, job-unit dependencies must be addressed, too. The latter occur, if processes of disjunctive job-units exchange messages. Saving communication channel content is required only in terms of coordinated checkpointing for reliable channels. In-transit messages of unreliable channels can be lost anyway.

Most kernel checkpointers do not support checkpointing communication channels or use different approaches. Therefore, XtreemGCP executes a common channel snapshot protocol that is integrated into the individual checkpointing sequence of each kernel checkpointer without modifying it.

XtreemGCP drains reliable communication-channels before taking a job-unit snapshot similiar to [15]. Therefore, we intercept *send* and *receive* socket functions. When the checkpointing callback function is executed the draining protocol is executed by creating a new draining thread after other functions optionally defined by the application programmer have finished. Of course we cannot just stop application threads as messages in transit need to be received first. In order to calculate how many bytes are in transit we count the number of sent and received bytes for each application thread on each grid node. This data can be accessed using the distributed AEM communication infrastructure. The draining thread sets a warning variable causing the interceptor to buffer all outgoing messages of involved threads when calling *send*. All these messages will not be passed to the network. Threads that need to receive in-transit data are allowed to run until they have received all in-transit messages. In the latter case a thread is stopped. Depending on the communication pattern it may take some time to drain all communication channels but by buffering outgoing data the protocol will eventually stop.

During restart all buffered messages need to be injected again. Flushing these buffers is done when restart callback functions are executed.

5 Grid Related Issues

5.1 Job Submission

In XtreemOS, the resource requirements of a job are described using the job submission description language (JSSDL). We have extended the JSSDL file of a job by checkpointing requirements which are defined in a separate XML file referenced. This extension allows the user/administrator to control checkpointing

for specific applications, e.g. the best suited checkpointing strategy can be listed including strategy parameters such as checkpoint frequency or number of checkpoints to be kept. Furthermore, it includes the selection of an appropriate kernel checkpointer that is capable of consistently checkpointing application resources e.g. multi-threaded processes, that are listed in the XML file. The discovery of a suitable kernel checkpointer is integrated in the resource discovery of AEM.

5.2 Checkpoint File Management

A job checkpoint is made up of job-unit checkpoint images and grid meta-data describing the checkpoint. Storing *checkpoint image files* can be costly due to their size and thus resource consumption. *Garbage collection* (GC) is essential to use disk space efficiently. The JCP implements the GC deleting checkpoint files not needed anymore. When checkpointing is used to migrate or suspend a job, job-unit checkpoint images are removed immediately after the job has been restarted. In these cases, the disk space needed to store the checkpoints is provided by the system. The system allows the user to define how many checkpoints to keep. However, the user has to provide the necessary disk space. If his quota is exhausted old checkpoints will be deleted. Candidates need to be calculated by the GC respecting constraints by incremental images and/or uncoordinated checkpointing (here a set of consistent checkpoints needs to be computed by the GC in order to detect unnecessary ones). *Grid-checkpointing meta-data* describe job and job-unit related data and include: the kernel checkpointers used, the checkpoint version number, the process list, the process group reference id, the process group reference type etc. The JCP relies on this meta-data during restart, e.g. for re-allocating resources valid before checkpointing, using the appropriate kernel checkpointers and for correctly addressing checkpoint images. In addition this meta-data is also useful as a statistical input for the adaptive checkpoint policy management.

5.3 Resource Conflicts

Identifiers of resources, e.g. process IDs, being valid before checkpointing must be available after restart to seamlessly proceed with application execution. Therefore, identifiers are saved in the meta-data and their availability is checked during restart by the responsible translation library. If an ID is not available a job unit must be migrated to another node.

To avoid these resource conflicts during restart we are currently integrating a light-weight virtualization (namespaces, e.g. applied to process IDs) and cgroups in order to isolate resources belonging to different applications. Now the kernel is able to distinguish equally named identifiers used by different jobs.

5.4 Security

Only authorized users should be able to checkpoint and restart a job. XtreemGCP relies on the XtreemOS security services to authenticate and authorize users [1].

Checkpoint/restart can introduce some issues regarding security because things may have changed between checkpoint and restart. For example, a library that was used by the job, and that was possibly checkpointed with the job, may have been updated to correct a security hole. The job owner may have lost some capabilities and is not allowed anymore to access some resources the job was using before restart, e.g. files. We cannot detail all possible cases here, but the general rule XtreemGCP follows is that transparency comes after security. Thus a job restart can fail because of security reasons.

6 Evaluation

In this section we discuss a preliminary evaluation of our architecture including the duration of a grid checkpoint/restart action and characteristic sequence calls across all services and layers of the XtreemGCP architecture. Here we do not evaluate scalability regarding number of nodes; this will of course depend on the application and the checkpointing strategy used.

The testbed nodes have Intel Core 2 Duo E6850 CPUs (3 GHz) with 2 GB DDR2-RAM and can be booted with either LinuxSSI 0.9.3 or a Linux kernel 2.6.20. For testing grid checkpoint/restart in connection with BLCR v0.8.0, one node loads the Linux image and executes AEM. For the LinuxSSI tests we use two nodes loading the LinuxSSI 0.9.3 image, one of them executing AEM.

Each checkpoint/restart sequence is splitted into several phases. The phases are executed within a new process owned by the appropriate XtreemOS user instead of root, thus, executing with correct users access rights. Each phase is controlled in Java at the grid-level using the Java Native Interface (JNI) to access the translation library and the kernel checkpointers.

The time values (in milliseconds) of each checkpoint/restart phase are shown in Table 1. These are average values taken from ten checkpoints/restarts measurements. The test application is a single-process application allocating five mega byte of memory space and sequentially writes a random value at each byte address. Such a simple application is sufficient to evaluate the overhead introduced by our architecture.

The native BLCR checkpointing and restart functionality is split into the appropriate XtreemGCP phases by controlling the kernel part of BLCR from the user-mode translation library using message queues.

Table 1. Duration of grid checkpointing and restart in milliseconds

Checkpoint					
	prepare	stop	checkpoint	resume	total
LinuxSSI (v0.9.3)	495,8	13,9	69,6	11,0	590,3
BLCR (v0.8.0)	381,2	40,1	250,5	5,3	677,1
Restart					
			rebuild	resume	total
LinuxSSI (v0.9.3)			2597,7	12,6	2610,3
BLCR (v0.8.0)			1659,3	5,7	1665,0

The **prepare** phase includes user authentication, determination of the process group reference id and type, saving the extracted meta-data to disk and the setup of phase control structures.

Within the **stop** phase ids passed by AEM will be filtered to remove non-job processes, e.g. the XtreemOS security process. The process group reference id is updated in case of intermediate changes.

The **checkpoint** call includes saving the process image to disk. This is the work done by native system-specific checkpointers. The time consumed here depends on the application behaviour, e.g. memory and file usage.

The **checkpoint resume** and **restart resume** phase resumes each process to proceed with execution. The first includes again process filtering for BLCR and LinuxSSI whereas the latter does not. Therefore, checkpoint resume is slower for both cases. The restart resume for LinuxSSI is faster than the one for BLCR because it does not require message-queue-based communication. All cases also include the release of the phase control structures.

The **rebuild** phase includes reading in meta-data, the setup of phase control structures, authentication and rebuilding of system-dependent kernel structures.

The observed overhead caused by XtreemGCP architecture is in the range of 0.5 - 2.6 seconds but independent of the number of nodes (it is a local overhead, only). Due to limited space we cannot discuss all implementation details causing minor time differences. Anyway, the times measured are rather small compared with the times needed to checkpoint large grid applications. For the latter the checkpoint phase dominates and may take minutes or even hours. Future work optimizations include a more fine-grained control of contents that really need to be saved and those which can be skipped (e.g. certain library files).

7 Related Work

XtreemGCP is the first implementation being able to checkpoint and restart distributed and parallel grid applications using heterogeneous kernel checkpointers.

The CoreGRID grid checkpointing architecture (GCA) [9] proposes a centralized solution to enable the use of low-level checkpointers. The *Grid Checkpoint Service* is the central component of the service that manages checkpoints in the grid and interacts with other grid services. The CGA implementation prefers the use of Virtual Machines (VMs) instead of kernel checkpointers. Overall it remains open to which extent CGA supports different kernel checkpointers and it is also currently not supporting checkpointing distributed applications.

The Open Grid Forum GridCPR Working Group targets application-level checkpointing [16][2]. For the latter they have described use-cases and requirements regarding APIs and related services. A service architecture comparable to the XtreemGCP architecture was deduced with a generic API partially included in SAGA. This work is hard to compare with our common kernel checkpoint API since it is limited to application-level checkpointing. But we plan to synchronize our common kernel checkpoint API with the one designed by OGF.

Adaptive checkpointing is important within dynamic grids and has been evaluated for parallel processing systems over peer-to-peer networks [12].

XtreemGCP callbacks and channel flushing mechanisms are inspired by previous work done by LAM/MPI [5]. The Open MPI Checkpoint/Restart framework [7] is very similar to XtreemGCP but limited to the MPI context. Moreover, since they do not especially target grids, they do not deal with grid specific issues.

8 Conclusion and Future Work

In this paper we have described the design and implementation of the XtreemGCP service used for job migration and fault tolerance. By introducing a common kernel checkpointer API, we are able to integrate and access different existing checkpointers in a uniform way. The proposed architecture is open to support different checkpointing strategies that may be adapted according to evolving failure situations or changing application behaviour.

We have also discussed challenges related to the implementation of the common kernel checkpointer API. We have described how to bridge different process group management techniques and how to save communication channels in a heterogeneous setup. Furthermore, we have introduced a callback mechanism for generic system-related checkpointing tasks and also for programmer-assisted optimizations. Other grid-related issues, e.g. checkpoint file management, security, and resource conflicts have been discussed, too.

The current prototype supports checkpointing and restarting jobs using BCLR and LinuxSSI checkpointers. Preliminary measurements show only minimal overhead introduced by the XtreemGCP architecture. XtreemOS source code is available at [1].

Future work includes optimizations, adaptive checkpointing, more measurements, and integration of the announced Linux kernel checkpointer.

References

1. <http://www.xtreemos.eu>
2. Badia, R., Hood, R., Kielmann, T., Merzky, A., Morin, C., Pickles, S., Sgaravatto, M., Stodghill, P., Stone, N., Yeom, H.: Use-Cases and Requirements for Grid Checkpoint and Recovery. Technical Report GFD-I.92, Open Grid Forum (2004)
3. Bhattiprolu, S., Biederman, E.W., Hallyn, S., Lezcano, D.: Virtual servers and checkpoint/restart in mainstream linux. SIGOPS Operating Systems Review 42(5), 104–113 (2008)
4. Coti, C., Herault, T., Lemarinier, P., Pilard, L., Rezmerita, A., Rodriguez, E., Cappello, F.: Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant mpi. In: SC 2006: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, Tampa, USA (2006)
5. Foster, I., Kesselman, C.: The globus project: a status report. Future Generation Computer Systems 15(5-6), 607–621 (1999)

6. Hupfeld, F., Cortes, T., Kolbeck, B., Focht, E., Hess, M., Malo, J., Marti, J., Stender, J., Cesario, E.: Xtreemfs - a case for object-based file systems in grids. *Concurrency and Computation: Practice and Experience* 20(8) (2008)
7. Hursey, J., Squyres, J.M., Mattox, T.I., Lumsdaine, A.: The Design and Implementation of Checkpoint/Restart Process Fault Tolerance for Open MPI. In: Int'l Parallel and Distributed Processing Symposium, Long Beach, CA, USA (2007)
8. Duell, J., Hargrove, P., Roman, E.: The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart. Technical Report LBNL-54941 (2003)
9. Jankowski, G., Januszewski, R., Mikolajczak, R., Stroinski, M., Kovacs, J., Kertesz, A.: Grid checkpointing architecture - integration of low-level checkpointing capabilities with grid. Technical Report TR-0075, CoreGRID, May 22 (2007)
10. Litzkow, M., Solomon, M.: The evolution of condor checkpointing. In: Mobility: processes, computers, and agents, pp. 163–164 (1999)
11. Mehnert-Spahn, J., Schoettner, M., Morin, C.: Checkpoint process groups in a grid environment. In: Int'l Conference on Parallel and Distributed Computing, Applications and Technologies, Dunedin, New Zealand (December 2008)
12. Ni, L., Harwood, A.: An adaptive checkpointing scheme for peer-to-peer based volunteer computing work flows. ArXiv e-prints (November 2007)
13. Plank, J.S., Beck, M., Kingsley, G., Li, K.: Libckpt: Transparent Checkpointing under Unix. In: Proceedings of USENIX Winter 1995 Technical Conference, New Orleans, Louisiana, USA, January 1995, pp. 213–224 (1995)
14. Reiser, H.P., Kapitza, R., Domaschka, J., Hauck, F.J.: Fault-tolerant replication based on fragmented objects. In: Eliassen, F., Montresor, A. (eds.) DAIS 2006. LNCS, vol. 4025, pp. 256–271. Springer, Heidelberg (2006)
15. Sankaran, S., Squyres, J.M., Barrett, B., Sahay, V., Lumsdaine, A., Duell, J., Hargrove, P., Roman, E.: The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing. *International Journal of High Performance Computing Applications* 19(4), 479–493 (2005)
16. Stone, N., Simmel, D., Kielmann, T., Merzky, A.: An Architecture for Grid Checkpoint and Recovery Services. Technical Report GFD-I.93, OGF (2007)
17. Vallée, G., Lottiaux, R., Rilling, L., Berthou, J.-Y., Dutka-Malhen, I., Morin, C.: A Case for Single System Image Cluster Operating Systems: the Kerrighed Approach. *Parallel Processing Letters* 13(2) (June 2003)

Scalable Transactions for Web Applications in the Cloud

Zhou Wei^{1,2,*}, Guillaume Pierre¹, and Chi-Hung Chi²

¹ Vrije Universiteit, Amsterdam, The Netherlands

zhouw@few.vu.nl, gpierre@cs.vu.nl

² Tsinghua University, Beijing, China

chichihung@mail.tsinghua.edu.cn

Abstract. Cloud computing platforms provide scalability and high availability properties for web applications but they sacrifice data consistency at the same time. However, many applications cannot afford any data inconsistency. We present a scalable transaction manager for cloud database services to execute ACID transactions of web applications, even in the presence of server failures. We demonstrate the scalability of our system using a prototype implementation, and show that it scales linearly to at least 40 nodes sustaining a maximum throughput of 7286 transactions per second.

1 Introduction

Cloud computing offers the vision of a virtually infinite pool of computing, storage and networking resources where applications can be scalably deployed [1]. The scalability and high availability properties of Cloud platforms however come at a cost. First, the scalable database services offered by the cloud such as Amazon SimpleDB and Google BigTable allow data query only by primary key rather than supporting secondary-key or join queries [2][3]. Second, these services provide only *eventual data consistency*: any data update becomes visible after a finite but undeterministic amount of time. As weak as this consistency property may seem, it does allow to build a wide range of useful applications, as demonstrated by the commercial success of Cloud computing platforms. However, many other applications such as payment services and online auction services cannot afford any data inconsistency. While primary-key-only data access is a relatively minor inconvenience that can often be accommodated by good data structures, it is essential to provide transactional data consistency to support the applications that need it.

A transaction is a set of queries to be executed atomically on a single consistent view of a database. The main challenge to support transactional guarantees in a cloud computing environment is to provide the ACID properties of

* This work is partially supported by the 863 project #2007AA01- Z122 & project #2008AA01Z12, the National Natural Science Foundation of China Project #90604028, and 973 project #2004CB719406.

Atomicity, Consistency, Isolation and Durability [4] without compromising the scalability properties of the cloud. However, the underlying data storage services provide only eventual consistency. We address this problem by creating a secondary copy of the application data in the transaction managers that handle consistency. Obviously, any centralized transaction manager would face two scalability problems: 1) A single transaction manager must execute all incoming transactions and would eventually become the performance bottleneck; 2) A single transaction manager must maintain a copy of all data accessed by transactions and would eventually run out of storage space. To support transactions in a scalable fashion, we propose to split the transaction manager into any number of Local Transaction Managers (LTMs) and to partition the application data and the load of transaction processing across LTMs.

Our system exploits two properties typical of Web applications to allow efficient and scalable operations. First, we observe that in Web applications, all transactions are short-lived because each transaction is encapsulated in the processing of a particular request from a user. This rules out long-lived transactions that make scalable transactional systems so difficult to design, even in medium-scale environments [5]. Second, Web applications tend to issue transactions that span a relatively small number of well-identified data items. This means that the two-phase commit protocol for any given transaction can be confined to a relatively small number of servers holding the accessed data items. It also implies a low (although not negligible) number of conflicts between multiple transactions concurrently trying to read/write the same data items.

A transactional system must maintain the ACID properties even in the case of server failures. For this, we replicate data items and transaction states to multiple LTMs, and periodically checkpoint consistent data snapshots to the cloud storage service. Consistency correctness relies on the eventual consistency and high availability properties of Cloud computing storage services: we need not worry about data loss or unavailability after a data update has been issued to the storage service. We assume a fail-stop failure model of the LTMs and do not address Byzantine failures.

It should be noted that the CAP dilemma proves that it is impossible to provide both strong Consistency and high Availability in the presence of network Partitions [6]. Typical cloud services explicitly choose high availability over strong consistency. In this paper, we make the opposite choice and prefer providing transactional consistency for the applications that require it, possibly at the cost of unavailability during network failures.

We demonstrate the scalability of our transactional database service using a prototype implementation. Following the data model of Bigtable, transactions are allowed to access any number of data items by primary key at the granularity of the data row. The list of primary-keys accessed by the transaction must be given before executing the transaction. This means for example that range queries are not supported. Our system supports both read-write and read-only transactions. We evaluate the performance of our prototype under a workload derived from the TPC-W e-commerce benchmark [7], and show that it scales

linearly to 40 LTMs sustaining a maximum throughput of 7286 transactions per second. This means that, according to the principles of Cloud computing, any increase in workload can be accommodated by provisioning more servers. Our system also tolerates server failures, which only cause a few aborted transactions (authorized by the ACID properties) and a temporary drop of throughput during data reorganization.

This paper is structured as follows. Section 2 presents related work. Then, Section 3 presents the design of partitioned transaction manager, and shows how to guarantee ACID properties even in the case of server failures. Section 4 presents performance evaluations and Section 5 concludes.

2 Related Work

Current prominent cloud database services such as Amazon SimpleDB and Google Bigtable only support eventual consistency properties [23]. To obtain a full database service on the cloud, one can easily deploy classical relational databases such as MySQL and Oracle, and thus get support for transactional consistency. However, these database systems rely on traditional replication techniques and therefore do not bring extra scalability improvement compared to a non-cloud deployment [8].

An alternative approach is to run any number of database engines in the cloud, and use the cloud storage service as shared storage medium [9]. Each engine has access to the full data set and therefore can support any form of SQL queries. On the other hand, this approach cannot provide full ACID properties. In particular, the authors claim that the Isolation property cannot be provided, and that only reduced levels of consistency can be offered.

The most similar system to ours is the Scalaris transactional DHT [10,11]. Like us, it splits data across any number of DHT nodes, and supports transactional access to any set of data items addressed by primary key. However, each query requires one or more requests to be routed through the DHT, adding latency and overhead. Cloud computing environment can also be expected to be much more reliable than typical peer-to-peer systems, which allows us to use more lightweight mechanisms for fault tolerance.

3 System Design

Figure 1 shows the organization of our transactional system. Clients issue HTTP requests to a Web application, which in turn issues transactions to a Transaction Processing System (TPS). The TPS is composed of any number of LTMs, each of which is responsible for a subset of all data items. The Web application can submit a transaction to any LTM that is responsible for one of the accessed data items. This LTM then acts as the coordinator of the transaction across all LTMs in charge of the data items accessed by the transaction. The LTMs operate on an in-memory copy of the data items loaded from the cloud storage service.

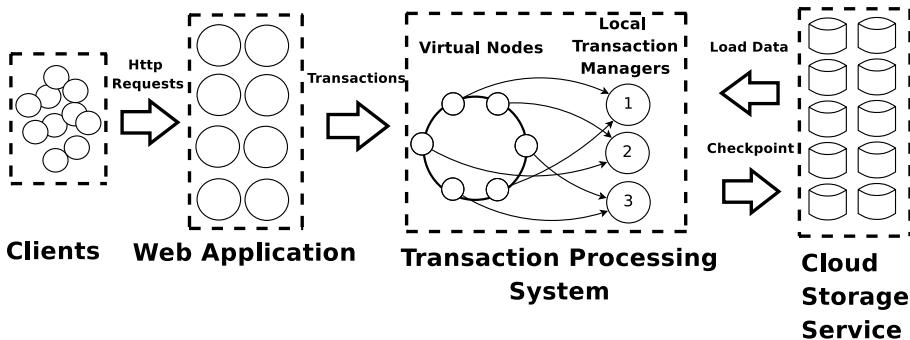


Fig. 1. System model

Data updates resulting from transactions are kept in memory of the LTM and periodically checkpointed back to the cloud storage service.

We implement transactions using the two-phase commit protocol. In the first phase, the coordinator requests all involved LTM and asks them to check that the operation can indeed be executed correctly. If all LTM vote favorably, then the second phase actually commits the transaction. Otherwise, the transaction is aborted.

We assign data items to LTM using consistent hashing [12]. To achieve a balanced assignment, we first cluster data items into virtual nodes, and then assign virtual nodes to LTM. As shown in Figure 1, multiple virtual nodes can be assigned to the same LTM. To tolerate LTM failures, virtual nodes and transaction states are replicated to one or more LTM. After an LTM server failure, the latest updates can then be recovered and affected transactions can continue execution while satisfying ACID properties.

We now detail the design of the TPS to guarantee the Atomicity, Consistency, Isolation and Durability properties of transactions. Each of the properties is discussed individually.

3.1 Atomicity

The Atomicity property requires that either all operations of a transaction complete successfully, or none of them do. To ensure Atomicity, for each transaction issued, our system performs two-phase commit (2PC) across all the LTM responsible for the data items accessed. If an agreement of “COMMIT” is reached, the transaction coordinator can return the result to the web application without waiting for the completion of the second phase [13].

To ensure Atomicity in the presence of server failures, all transaction states and data items should be replicated to one or more LTM. When an LTM fails, the transactions it was coordinating can be in two states. If a transaction has reached an agreement to “COMMIT,” then it must eventually be committed; otherwise, the transaction can be aborted. Therefore, we replicate transaction states in two occasions: 1) When an LTM receives a new transaction, it must

replicate the transaction state to other LTMs before confirming to the application that the transaction has been successfully submitted; 2) After all participant LTMs reach an agreement to “COMMIT” at the coordinator, the coordinator updates the transaction state at its backups. This creates in essence in-memory “redo logs” at the backup LTMs. The coordinator must finish this step before carrying out the second phase of the commit protocol. If the coordinator fails after this step, the backup LTM can then complete the second phase of the commit protocol. Otherwise, it can simply abort the transaction without violating the Atomicity property.

An LTM server failure also results in the inaccessibility of the data items it was responsible for. It is therefore necessary to re-replicate these data items to maintain N replicas. Once an LTM failure is detected, the failure detector issues a report to all LTMs so that they can carry out the recovery process and create a new consistent membership of the system. All incoming transactions that accessed the failed LTM are aborted during the recovery process. If a second LTM server failure happens during the recovery process of a previous LTM server failure, the system initiates the recovery of the second failure after the current recovery process has completed. The transactions that cannot recover from the first failure because they also accessed the second failed LTM are left untouched until the second recovery process.

As each transaction and data item has $N + 1$ replicas in total, the TPS can thus guarantee the Atomicity property under the simultaneous failure of N LTM servers.

3.2 Consistency

The Consistency property requires that a transaction, which executes on a database that is internally consistent, will leave the database in an internally consistent state. Consistency is typically expressed as a set of declarative integrity constraints. We assume that the consistency rule is applied within the logic of transactions. Therefore, the Consistency property is satisfied as long as all transactions are executed correctly.

3.3 Isolation

The Isolation property requires that the behavior of a transaction is not impacted by the presence of other transactions that may be accessing the same data items concurrently. In the TPS, we decompose a transaction into a number of sub-transactions. Thus the Isolation property requires that if two transactions conflict on more than one data item, all of their conflicting sub-transactions must be executed sequentially, even though the sub-transactions are executed in multiple LTMs.

We apply timestamp ordering for globally ordering conflicting transactions across all LTMs. Each transaction has a globally unique timestamp, which is monotonically increasing with the time the transaction was submitted. All LTMs then order transactions as follows: a sub-transaction can execute only after all

conflicting sub-transactions with a lower timestamp have committed. It may happen that a transaction is delayed (e.g., because of network delays) and that a conflicting sub-transaction with a younger timestamp has already committed. In this case, the older transaction should abort, obtain a new timestamp and restart the execution of all of its sub-transactions.

As each sub-transaction accesses only one data item by primary key, the implementation is straightforward. Each LTM maintains a list of sub-transactions for each data item it handles. The list is ordered by timestamp so LTMs can execute the sub-transactions sequentially in the timestamp order. The exception discussed before happens when an LTM inserts a sub-transaction to the list but finds its timestamp smaller than the one currently being executed. It then reports the exception to the coordinator LTM of this transaction so that the whole transaction can be restarted. We extended the 2PC with an optional “RESTART” phase, which is triggered if any of the sub-transactions reports an ordering exception. After a transaction reached an agreement and enters the second phase of 2PC, it cannot be restarted any more.

We are well aware that assigning timestamps to transactions using a single global timestamp manager can create a potential bottleneck in the system. We used this implementation for simplicity, although distributed timestamp managers exist [14].

3.4 Durability

The Durability property requires that the effects of committed transactions would not be undone and would survive server failures. In our case, it means that all the data updates of committed transactions must be successfully written back to the backend cloud storage service.

The main issue here is to support LTM failures without losing data. For performance reasons, the commit of a transaction does not directly update data in the cloud storage service but only updates the in-memory copy of data items in the LTMs. Instead, each LTM issues periodic updates to the cloud storage service. During the time between a transaction commit and the next checkpoint, durability is ensured by the replication of data items across several LTMs. After checkpoint, we can rely on the high availability and eventual consistency properties of the cloud storage service for durability.

When an LTM server fails, all the data items stored in its memory that were not checkpointed yet are lost. However, as discussed in Section 3.1, all data items of the failed LTM can be recovered from the backup LTMs. The difficulty here is that the backups do not know which data items have already been checkpointed. One solution would be to checkpoint all recovered data items. However, this can cause a lot of unnecessary writes. One optimization is to record the latest checkpointed transaction timestamp of each data item and replicate these timestamps to the backup LTMs. We further cluster transactions into groups, then replicate timestamps only after a whole group of transactions has completed.

Another issue related to checkpointing is to avoid degrading the system performance at the time of a checkpoint. The checkpoint process must iterate through the latest updates of committed transactions and select the data items to be checkpointed. A naive implementation that would lock the whole buffer during checkpointing would also block the concurrent execution of transactions. We address this problem by maintaining a buffer in memory with the list of data items to be checkpointed. Transactions write to this buffer by sending updates to an unbounded non-blocking concurrent queue [15]. This data structure has the property of allowing multiple threads to write concurrently to the queue without blocking each other. Moreover, it orders elements in FIFO order, so old updates will not override younger ones.

3.5 ReadOnly Transactions

Our system supports read-write and read-only transactions indifferently. The only difference is that in read-only transactions no data item is updated during the second phase of 2PC. Read-only transactions have the same strong data consistency property, but also the same constraint: accessing well identified data items by primary key only. However, our system provides an additional feature for read-only transactions to support complex read queries such as range queries executed on a consistent snapshot of database.

We exploit the fact that many read queries can produce useful results by accessing an older but consistent data snapshot. For example, in e-commerce Web applications, a promotion service may identify the best seller items by aggregating recent orders information. However, it is not necessary to compute the result based on the absolute most recent orders. We therefore introduce the concept of Weakly-Consistent Read-only Transaction (WCRT), which is defined as follows: 1) A WCRT allows any type of read operations, including range queries; 2) WCRTs do not execute at the LTM but access the latest checkpoint in the cloud storage service directly. A WCRT always executes on an internally consistent but possibly slightly outdated snapshot of the database. WCRTs are supported only if the underlying data storage service supports multi-versioning, as for example Bigtable [3]. To implement WCRTs, we introduce a snapshot mechanism in the checkpoint process, which marks each data update with a specific snapshot ID that is monotonically increasing. This ID is used as the version number of the new created version when it is written to the cloud storage service. A WCRT can thus access a specific snapshot by only reading the latest version of any data item of which the timestamp is not larger than the snapshot ID.

We define that the updates of every M transactions with subsequent timestamps constitute a new snapshot. Assuming that the transaction timestamp is implemented as a simple counter, the first snapshot reflects all the updates of committed transactions $[0, M]$. The next snapshot reflects updates from transactions $[0, 2M]$, and so on. A snapshot is “ready” after all the updates of transactions it reflects have been written back to cloud storage service and have been made visible to the Web application. The TPS publishes the latest snapshot that is “ready,” so the web application can access the consistent view of data from cloud storage service.

The difficulty in implementing the snapshot mechanism is that each LTM performs checkpoints independently and that any transaction may update data items in multiple LTMs. We introduce a Master node which collects reports from each LTMs about its latest *locally* “ready” snapshot. So the Master node can determine the latest *globally* “ready” snapshot. This Master node also carries out the task of publishing the latest snapshot ID to Web applications. If the underlying cloud data storage service satisfies “Read-Your-Writes” consistency, writing back successfully will be a sufficient condition to consider the snapshot as “ready.” Otherwise, the LTMs must check if the issued updates is readable.

4 Evaluation

We demonstrate the scalability of our transactional database system by presenting the performance evaluation of a prototype implementation under the workload of TPC-W [7]. TPC-W is an industry standard e-commerce benchmark that models an online bookstore similar to Amazon.com. This paper focuses on the scalability of the system with increasing number of LTMs rather than on its absolute performance for a given number of LTMs. Our evaluation assumes that the application load remains roughly constant and studies the scalability in terms of the maximum sustainable throughput under a response time constraint.

4.1 Experiment Setup

All experiments are performed on the DAS-3 at the Vrije Universiteit, an 85-node Linux-based server cluster [16]. Each machine in the cluster has a dual-CPU / dual-core 2.4 GHz AMD Opteron DP 280, 4 GB of memory and a 250 GB IDE hard drive. Nodes are connected to each other with a Myri-10G LAN such that the network latency between the servers is negligible. We use Tomcat v5.5.20 as application server and HBase v0.2.1, an open-source clone of BigTable [17] as backend cloud storage service. The TPC-W application and load generator are deployed in separate application servers.

We generate an evaluation workload composed of transactions issued by the TPC-W Web application. The workload is generated by a configurable number of Emulated Browsers (EBs), each of which issues requests from a simulated user. We observe that most of the read queries and read-only transactions of TPC-W can tolerate a slightly old version of data, so they can directly access a consistent snapshot from the storage service using WCRTs. The workload that an Emulated Browser issues to the TPS mainly consists of read-write transactions that require strong data consistency. Figure 2 shows the workflow of transactions issued by an Emulated Browser, which simulates a typical shopping process of customer. Each EB waits for 500 milliseconds between receiving a response and issuing the next transaction.

We adapted the original relational data model defined by TPC-W to the Bigtable data model, so that the application data can be stored into HBase. Using similar data denormalization techniques as in [18], we designed a Bigtable data model for

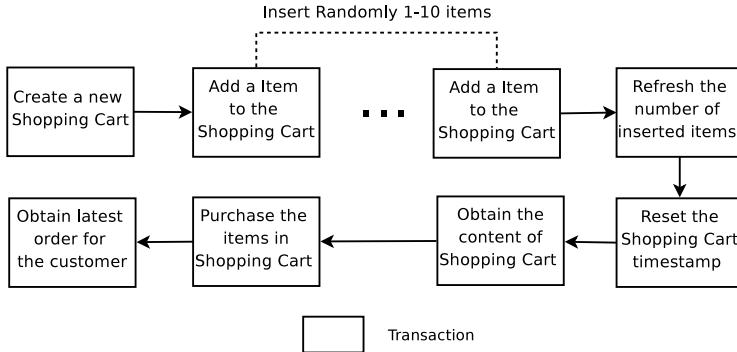


Fig. 2. Workflow of transactions issued by each Emulated Browser (EB) of TPC-W

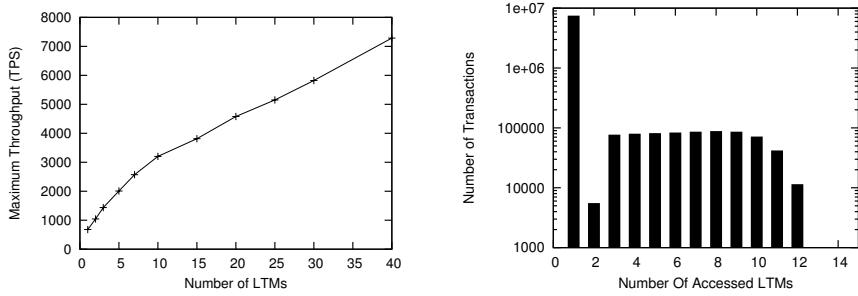
TPC-W that contains only the columns accessed by the transactions in Figure 2. The relational data model of TPC-W comprises six tables that are accessed by these transactions. To adapt this data model to Bigtable, we first combine five of them (“Orders, Order_Line, Shopping_Cart, Shopping_Cart_Entry, CC_XACTS”) into one *bigtable* named “Shopping.” Each of the original tables is stored as a column family. The new *bigtable* “Shopping” has the same primary key as table “Shopping_Cart.” For table “Order_Line,” multiple rows are related to one row in table “Order,” they are combined into one row and stored in the new *bigtable* by defining different column names for the values of same data column but different rows. Second, for the remaining table “Item,” only the column “i_stock” is accessed. We thus can have a *bigtable* named “Item_Stock” which only contains this column and has the same primary key. Finally, for the last transaction in Figure 2 which retrieves the latest order information for a specific customer, we create an extra index *bigtable* “Latest_Order” which uses customer IDs as its primary key and contains one column storing the latest order ID of the customer.

Before each experiment, we populate 144,000 customer records in the “Latest_Order” *bigtable* and 10,000 item records in the “Item_Stock” *bigtable*. We then populate the “Shopping” *bigtable* according to the benchmark requirements.

In our experiments, we observed a load balancing problem in HBase because TPC-W assigns new shopping cart IDs sequentially. Each HBase node is responsible for a set of contiguous ranges of ID values, so at any moment of time, most newly created shopping carts would be handled by the same HBase node. To address this problem, we horizontally partitioned the *bigtables* into 50 sub-*bigtables* and allocated data items in round-robin fashion.

4.2 Scalability Evaluation

We study the scalability of the Transaction Processing System by measuring the system performance in terms of throughput. We first define a response time constraint that imposes that the 99% of transactions must return within 100 ms. For a given number of LTM_s we then measure the maximum sustainable throughput of the system before the constraint gets violated.



(a) Maximum Sustainable Throughput

(b) Number of LTMs accessed by the transactions of TPC-W

Fig. 3. Throughput Scalability of the Transaction Processing System

We configure the system so that each transaction and data item has one backup in total, and set the checkpoint interval to 1 second. We start with one LTM and 5 HBase servers, then add more LTM and HBase servers. In all cases, we deliberately over-allocated the number of HBase servers and client machines to make sure that the Transaction Processing System remains the performance bottleneck.

Figure 3(a) shows that our system scales nearly linearly. When using 40 LTM servers it reaches a maximum throughput of 7286 transactions per second generated by 3825 emulated browsers. In this last configuration, we use 40 LTM servers, 36 HBase servers, 3 clients to generate load, 1 server as global timestamp manager, and 1 Master server for managing snapshots for WCRTs and coordinating the recovery process. This configuration uses the entire DAS-3 cluster so we could not extend the experiment further. The maximum throughput of the system at that point is approximately 10 times that of a single LTM server.

Figure 3(b) shows the number of LTMs that participate in the transactions¹, when there are 40 LTM servers in the system. Transactions access up to 12 LTMs, which corresponds to a shopping cart containing the maximum number of 10 items. We however observe that the vast majority of transactions access only one LTM. In other words, most of the transactions of TPC-W execute within one LTM and its backups only. We expect this behavior to be typical of Web applications.

4.3 Fault Tolerance

We now study the system performance in the presence of LTM server failures. We configure the system so that each transaction and data item has one backup and set the checkpoint interval to 1 second. We start with 5 LTMs and generate the workload using 500 EBs so that the system would not overload even after two LTM servers failures.

We first warm up the system by adding 25 EBs every 10 seconds. The full load is reached after 200 seconds. After running the system normally for a while, one LTM server is shutdown to simulate a failure. Figure 4(a) shows the effect

¹ The LTMs that act only as backup of transactions or data items are not counted in.

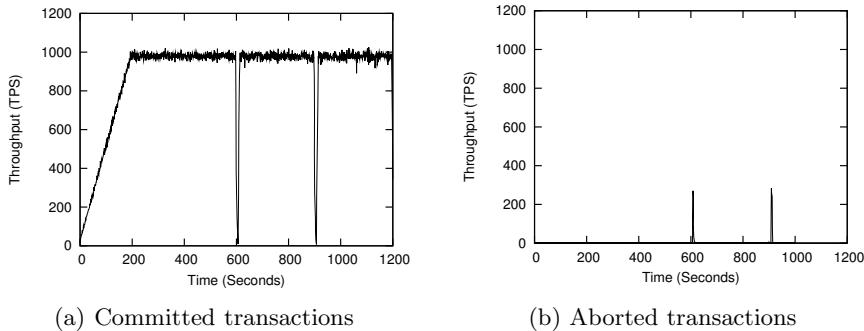


Fig. 4. Throughput of system in the presence of LTM server failures

of two LTM server failures at times 600 and 900 seconds. When an LTM server fails, the system recovers within a few seconds and the transaction throughput returns to the previous level. At the same time, as shown in Figure 4(b), a few transactions are aborted because the incoming transactions that accessed the failed LTM must be rejected during the recovery process. After two LTM server failures, the remaining 3 LTM servers can still sustain a throughput of about 1000 transactions per second.

5 Conclusion

Many Web applications need strong data consistency for their correct executions. However, although the high scalability and availability properties of the cloud make it a good platform to host Web content, scalable cloud database services only provide eventual consistency properties. This paper shows how one can support ACID transactions without compromising the scalability property of the cloud for web applications, even in the presence of server failures.

This work relies on few simple ideas. First, we load data from the cloud storage system into the transactional layer. Second, we split the data across any number of LTMs, and replicate them only for fault tolerance. Web applications typically access only a few partitions in any of their transactions, which gives our system linear scalability. Our system supports full ACID properties even in the presence of server failures, which only cause a temporary drop in throughput and a few aborted transactions.

Data partitioning also implies that transactions can only access data by primary key. Read-only transactions that require more complex data access can still be executed, but on a possibly outdated snapshot of the database. Lifting this limitation is on our immediate research agenda.

References

1. Hayes, B.: Cloud computing. *Communications of the ACM* 51(7), 9–11 (2008)
2. Amazon.com: Amazon SimpleDB, <http://aws.amazon.com/simpledb>

3. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: a distributed storage system for structured data. In: Proc. OSDI, pp. 205–218 (2006)
4. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann, San Francisco (1993)
5. Transaction Processing Performance Council: TPC benchmark C standard specification, revision 5 (December 2006), <http://www.tpc.org/tpcc/>
6. Gilbert, S., Lynch, N.: Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News 33(2), 51–59 (2002)
7. Smith, W.D.: TPC-W: Benchmarking an ecommerce solution. White paper, Transaction Processing Performance Council
8. Atwood, M.: A MySQL storage engine for AWS S3. In: MySQL Conference and Expo. (2007), <http://fallenpegasus.com/code/mysql-awss3/>
9. Brantner, M., Florescu, D., Graf, D., Kossmann, D., Kraska, T.: Building a database on S3. In: Proc. ACM SIGMOD, pp. 251–264 (2008)
10. Moser, M., Haridi, S.: Atomic Commitment in Transactional DHTs. In: Proc. CoreGRID Symposium (2007)
11. Plantikow, S., Reinefeld, A., Schintke, F.: Transactions for distributed wikis on structured overlays. In: Proc. Intl. Workshop on Distributed Systems: Operations and Management, pp. 256–267 (2007)
12. Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., Lewin, D.: Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In: Proc. ACM Symposium on Theory of Computing, pp. 654–663 (1997)
13. Hvasshovd, S.O., Torbjørnsen, O., Bratsberg, S.E., Holager, P.: The ClustRa Telecom Database: High Availability, High Throughput, and Real-Time Response. In: Proc. VLDB, pp. 469–477 (1995)
14. Raz, Y.: The principle of commitment ordering, or guaranteeing serializability in a heterogeneous environment of multiple autonomous resource managers using atomic commitment. In: Proc. VLDB, pp. 292–312 (1992)
15. Michael, M., Scott, M.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proc. ACM symposium on Principles of distributed computing, pp. 267–275 (1996)
16. DAS3: The Distributed ASCI Supercomputer 3, <http://www.cs.vu.nl/das3/>
17. HBase: An open-source, distributed, column-oriented store modeled after the Google Bigtable paper, <http://hadoop.apache.org/hbase/>
18. Wei, Z., Dejun, J., Pierre, G., Chi, C.-H., Steen, M.v.: Service-oriented data denormalization for scalable web applications. In: Proc. Intl. World Wide Web Conf. (2008)

Provider-Independent Use of the Cloud

Terence Harmer, Peter Wright, Christina Cunningham, and Ron Perrott

Belfast e-Science Centre, The Queen's University of Belfast, Belfast BT7 1NN, UK

{t.harmer, p.wright, c.cunningham, r.perrott}@besc.ac.uk

Abstract. Utility computing offers researchers and businesses the potential of significant cost-savings, making it possible for them to match the cost of their computing and storage to their demand for such resources. A utility compute provider enables the purchase of compute infrastructures on-demand; when a user requires computing resources a provider will provision a resource for them and charge them only for their period of use of that resource. There has been a significant growth in the number of cloud computing resource providers and each has a different resource usage model, application process and application programming interface (API)—developing generic multi-resource provider applications is thus difficult and time consuming. We have developed an abstraction layer that provides a single resource usage model, user authentication model and API for compute providers that enables cloud-provider neutral applications to be developed. In this paper we outline the issues in using external resource providers, give examples of using a number of the most popular cloud providers and provide examples of developing provider neutral applications. In addition, we discuss the development of the API to create a generic provisioning model based on a common architecture for cloud computing providers.

1 Introduction

Utility computing offers researchers and businesses the potential of significant cost-savings in that it is possible for them to match the cost of their computing and storage to their demand for such resources. Thus, a user needing 100 compute resources for 1 day per week may purchase those resources only for the day they are required. A utility provider may also enable the allocation of storage to hold data and network bandwidth for access to a user's applications and services. Again, this enables a user to match their storage and network capacity to meet their needs. More generally, it is possible for a business or researcher to work without owning any significant infrastructure and rely on providers when they need a compute infrastructure.

On-demand provision of resources to users has been around for some time with grid computing services, such as the UK National Grid Service^[1] and the US TeraGrid^[2], but they have largely been job focused rather than service-focused. In these infrastructures the goal is to optimise the compute jobs that are being requested by users—delivering the fastest compute for users and making the best use of the fixed compute infrastructure.

With the success of the cloud computing paradigm [3], dynamic infrastructures have become popular. As a result of the increased commercial demand, there are more providers offering infrastructure on-demand—such as Amazon EC2 [4], Flexiscale [5], AppNexus [6], NewServers [7] and ElasticHosts [8]. Sun have also announced plans to become a cloud provider/. Most vendors provide wholly virtual servers where a user provisions an infrastructure by providing a virtual machine (VM). Within a VM a user can bundle applications, services and data together in the same package. VMs usually give reduced performance when compared to the native compute resource but do enable the provider to share its resources in a secure and reliable manner. However, there are vendors that offer physical machines on-demand, such as NewServers, that are suited to high computation/IO applications.

1.1 Why Are Resource Providers Interesting to Us?

At the Belfast e-Science Centre, our interest is in creating dynamic service-focused infrastructures which are primarily in the media [9] [10] and finance [11] sectors. Media and finance are challenging domains to develop applications for in that:

- they require high computational throughput with millisecond-based quality of service—for example to create a particular video format for a user as they request to view it;
- they have high security requirements by the nature of the value of the data that is processed and the legislation that regulates the sectors—for example, in communicating the details of the ownership of shares; and
- they are subject to rapid peaks and troughs in demand—for example, in a volatile day trading volumes can double in minutes as traders react to news (and other trader behaviour!)

We have developed a framework of services that enables our applications to be deployed and configured on demand within a fixed infrastructure. In developing our application it quickly became clear that the peaks and troughs in demand could best be accommodated using on-demand providers to supplement a fixed infrastructure. More recently, we have created and deployed services for commercial and research partners where none of the infrastructure was owned by the user and instead relied entirely on cloud resource providers.

We have worked on extending our dynamic deployment services from supporting only owned and fixed compute resources to include resources allocated on demand by cloud resource providers. To do this our support software needed a way to provision resources, configure those resources, deploy software onto the resources and, when its task was complete, discard the allocated resources—all whilst providing accounting for the resources that were used. Investigation turned up no standard cloud API; the closest is the Eucalyptus [12] open source provider codebase, however it is a clone of the EC2 API. It does not seem likely that a standard model for cloud computing providers will appear soon given the speed at which cloud vendors are innovating and enhancing their services. In addition

(given the competition between providers) it seems unlikely at this stage that many providers would believe it in their interest to create or comply with a common API.

1.2 What Is Involved in Using an On-Demand Cloud Provider?

Cloud computing is an increasingly compelling prospect for researchers, start-ups and large organisations, either as a substitution for owned hardware or to bolster their existing infrastructure in high-load scenarios. Multiple providers are emerging, offering different features such as hardware load-balancing, physical (rather than virtual) resources, and private VLANs capable of multicast. These are attractive because of the low cost of use when compared to buying, refreshing and supporting a large infrastructure.

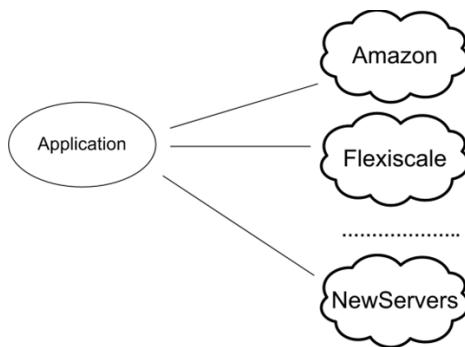


Fig. 1. A traditional multi-cloud application

It is in the interest of providers to have their own APIs as this simplifies their development task, fitting (perfectly) their business model and their implementation. However, these varying APIs complicate service clouds, such as our media and financial applications, that wish to sit on top of multiple cloud vendors that provide physical resources. Ideally, these applications should be written in a provider agnostic fashion using the cheapest resources when they are required and switching between providers when there are cost savings in doing so (Fig. 11). Unfortunately, the application must currently be aware of the provider being used, their utility model and their API. This makes developing applications that can use multiple providers difficult to write and maintain because you must be aware of the benefits and drawbacks of particular providers.

In addition, APIs are also evolving with new functionality, operating systems, SLA terms, pricing and resource specifications. Keeping up-to-date with the latest API changes can be time-consuming and can mean that applications are fixed to particular vendors and compute resources.

As an illustration of the complexities faced by various developers working in the cloud, we consider a simple and basic task in using a cloud provider – instantiate the cheapest machine available and display its instance identity and

IP address. We present Java code for 3 providers: Amazon EC2, Flexiscale and NewServers. (The APIs have been simplified for readability, however real use is nearly identical to what is presented here).

NewServers. NewServers provides on-demand access to native (*bare-metal*) compute resources. The simplicity and minimalism of their API is interesting to us however, since it provides a good model for small home-grown academic clouds.

```
String size = 'small'; // 2.8GHz Xeon, 1GB RAM, 36GB disk
String os   = 'centos5.2';

// create & start
Server server = newservers.addServer(size, os);

String instId = server.getId();
String ip     = server.getPublicIP();

System.out.printf('Instance %s has IP %s\n', instId, ip);
```

From their API we know that “small” is their most basic machine with a 2.8GHz Xeon and 1GB of RAM. We choose that, deploying a CentOS 5.2 image onto it. Once we call *addServer*, the server is created and started and its details are displayed. We can query their API for a list of available template machines and operating systems and their costs; we’ve elected not to do that in this case.

Flexiscale. Flexiscale is a cloud resource provider that supports deployment of a range of VM types. It provides near native network support capability: providing VLANs and multicast, for example, with high levels of VM quality of service and support for automatic re-deployment in the event of failure. The objective is to create an easy-to-use mechanism with near native machine capability and automated resilience.

```
String instId = 'MyNewServer'; // User-assigned id
// Get default hosting package, VLAN, and OS
Package fxPackage = flexiscale.listPackages().get(0);
Vlan fxVlan = flexiscale.listVlans().get(0);
OSImage os = new OSImage(27); // Ubuntu

Server s = new Server();
s.setName(instId);
s.setDisk_capacity(4); // 4GB disk
s.setProcessors(1); // 1 CPU
s.setMemory(512); // 512MB RAM
s.setPackage_id(fxPackage.id);
s.setOperatingSystem(os);
s.setInitialPassword('changeme');

flexiscale.createServer(s, fxVlan); // allocate
flexiscale.startServer(instId, 'no notes'); // start

String ip = flexiscale.listServers(instId)[0].getIP(0);
System.out.printf('Instance %s has IP %s\n', instId, ip);
```

In the above code we acquire our hosting package, default VLAN and a pointer to the Ubuntu system image. We create a new server with 1 CPU, 512MB RAM and 4GB local disk. Once created, we start the machine and then request its details from the API to determine the IP address. With Flexiscale we must know the valid CPU, RAM, OS combination—this is available in a table on their API site. With the Flexiscale API we can allocate our VMs to specific VLANs, enabling applications to have secure and good low-level control of communications. Flexiscale must create a machine image when requested and so turnaround time is about 10 minutes and often it is useful to create a resource in advance of when it might be required by an application. This can complicate scaling to meet changes in application demand.

Amazon EC2. Amazon is the largest and most widely known cloud compute provider that uses a proven infrastructure (Amazon’s own infrastructure) and enables rapid scaling of resources by a user. Amazon provides hosting capabilities in the US and in Europe.

```
RunInstancesRequest req = new RunInstancesRequest();
req.setImageId ('ami-1c5db975'); // Ubuntu
req.setKeyName ('someAuthKey'); // preloaded root ssh key
req.setPlacement ('us-east-1a'); // the datacentre
// 1GHz xeon, 1.7GB RAM, 150GB disk
req.setInstType ('m1.small');

// allocate & start
Reservation res = ec2.runInstances (req).getReservation ();

String id = res.getInstance ()[0].getInstanceId ();
String ip = res.getInstance ()[0].getPublicDNS ();

System.out.printf ('Instance %s has IP %s\n', id, ip);
```

With EC2, we specify the exact Amazon Machine Image for an Ubuntu operating system, an authentication key to secure SSH access, a datacentre (“us-east-1a”) to place our VM in, and the name of the cheapest template (“m1.small”). EC2 runs as a large compute cloud that allocates resources in different physical datacentres (EC2 sub-clouds). With EC2 you must initially pre-store a VM image with Amazon and record Amazon’s unique identity for that image. When deploying you must be careful to ensure that each VM compute resource you create is within the same datacentre sub cloud—this will improve performance but also avoids significant charges for inter-cloud bandwidth. It is also necessary to know the type of OS instance that is permissible—this is available from a table on their API site. EC2 boots VMs very quickly, giving a low turnaround time and enabling rapid scaling of infrastructure.

1.3 Lessons from Existing Cloud Provider APIs

As we can see from the previous examples, the initialisation and use of a resource in a resource provider varies significantly in implementation details. Some providers let you specify exact machine configurations while others have sets

of configurations to choose from when requesting a resource. Some allow user-assigned resource identifiers while others allocate an identifier for you. The usage models are different also. Some models have the concept of a “stopped machine”; others consider stopping to be analogous to discarding.

To create a generic, cloud-provider independent application, a simple consistent API is necessary that provides a consistent and flexible provider API and resource usage model. Such an API would allow users to specify requirements (such as RAM size, CPU performance, disk space) and enable a rich set of filters that match requirements with provider capabilities. In addition, the model would permit a consistent and simple resource model that attempts to hide the particular details of resource providers.

2 Our Abstraction Layer

Our need for an abstraction layer (Fig. 2) to provide a generic cloud provider was rooted in a number of our project applications which created highly dynamic service-based infrastructures [3] [9] [10] which

- were deployed and managed dynamically;
- used autonomic management services to control and monitor the infrastructure;
- that scaled to match user demand; and
- employed location-aware services to optimise user access and comply with data processing regulations.

Based on our experience of using cloud providers and their evolution over time (as well as where we see them going in the future) we formed an outline usage model which we see as common to them all:

1. Find viable compute resources (based on properties such on CPU speed, memory, disk capacity, etc.)
2. Select the best match from those available (based on specification/price and the trade-off from the options available from the resource vendors)
3. Configure compute resource parameters (such as OS, firewall, storage, etc)
4. Instantiate a box using that configuration

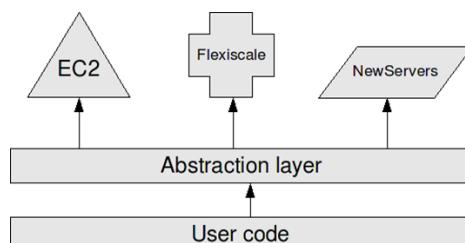


Fig. 2. Multi-cloud applications with our abstraction layer

5. User does some work on the box
6. Discard the compute resource

This model is intentionally simple and attempts to reflect an ideal for the application user who wishes to specify resources without knowing the internal behaviour of the resource provider. Currently, when developing cloud applications, *Step 1: Find viable compute resources* and *Step 2: Select the best match*, are performed manually by the development/deployment team in creating/deploying the application, limiting the extent to which an application can dynamically take advantage of new resource types and requiring the implementation team to refine an application to take advantage of benefits of new vendors or machine types/pricing.



Fig. 3. Simple management model

In essence, our workflow (Fig. 3), reflecting our usage model, is find, instantiate, manage and discard. There is an increased focus on predictability within the cloud provider community to encourage traditional business IT to use cloud technology. For example, Amazon has recently-announced a Service Level Agreement (SLA) to bring them in line with other cloud providers. These SLAs aim to provide, for example, a quantifiable up-time for their resources and penalty clauses if this is not met. In addition, a large-scale EU project, SLA@SOI [13], aims to make SLA negotiation a core part of its resource provisioning model with resource providers chosen to satisfy SLA requirements and being expected to enforce SLA requirements.



Fig. 4. Expanded management model

To allow for this concept of predictability we add a *reserve* stage (Fig. 4) to allow users to indicate their near-term interest in allocating some resource—that may incur a small cost with some providers. This smoothes out the differences between providers who do have the concept (e.g. Flexiscale) and those who do not (e.g. Amazon) and will in the future allow calls to enable the provider to keep ahead of demand by requiring reservation and potentially minimising the number of hosting machines switched on at any given time.

We see the role of the resource provider, once the resource has been allocated, as completely behind the scenes—communication and control of the allocated resource is identical to every other machine in the user’s datacentre. Thus, we

talk about the dynamic resources as “boxes” to indicate that the vendor is providing nothing more than a machine with an OS preinstalled that is configured according to the user’s specification.

So, with our model resource providers are called *Box Providers*; they offer various *Box Templates* that describe properties of the boxes they can allocate. These templates define the type of box, how it can be configured and define the pricing structures and usage policies for the box. The templates can then be queried by generic questions like “How much would 48 hours of uptime with 1GB of internet transfer up and 10GB down cost me?” or specific questions for particular providers “How much is 50GB of transfer to Amazon S3 going to cost me?”. The use of templates completely isolates programs from the variations in pricing and available resources, making the process naturally extensible. A user’s focus is on the type of resource they require with the allocation process finding a provider to satisfy the generic resource request and the non-functional requirements (such as price) that might be appropriate.

Once a template has been chosen it may be reserved with particular configuration properties—for example, setting the required OS, firewall rules, storage mounts, etc. At a later stage the reservation can then be instantiated, giving a *Box Instance* which can be managed, queried and finally discarded.

Allocating a Resource with Our Abstraction. Using our layer, the previous examples do not change much:

```
EC2Credentials account = ...;
BoxProvider provider = new EC2BoxProvider(account);

// Use the cheapest template this provider knows about
BoxTemplate templ = BoxTemplateHelper.cheapest(provider);

// Deploy 1 ubuntu box with quickstart defaults
BoxConfiguration os = BasicBoxConfig.getUbuntu(templ);
ReservedBox res = provider.reserve(templ, 1, os);

// Create & start the machine
BoxInstance inst = provider.instantiate(res);

String instId = inst.getId();
String ip = inst.getPublicIp();
System.out.printf('Instance %s has IP %s\n', instId, ip);

provider.discard(inst); // throw the machine away
```

As in our previous examples, the process is the same, however since we are expressing things generically it would be trivial to swap out our EC2BoxProvider for a FlexiscaleBoxProvider and, without any other code changes, allocate machines using Flexiscale. Given this view, we can consider a more interesting scenario – we want a 5-machine cluster, each with 1GB of RAM and 160GB of disk; we will be uploading a 1GB file for some light processing by our mini-cluster into a 10GB output file which will then be retrieved within 2 days. The calculation will be very integer-heavy so we’ll use a Java integer performance benchmark as a fitness function.

```

BoxProvider provider = new BoxProviderUnion(getEC2(),
                                             getFxscale(),
                                             ...);

// 1GB of RAM, 160GB disk
Restriction r1 = new MemoryRestriction(1024);
Restriction r2 = new LocalDiskRestriction(160);

BoxTemplate[] templates = provider.find(r1, r2);

// Find cheapest for 48 hours with 1GB up, 10GB down
int hoursOn = 48;
int mbUpload = 1024, mbDownload = 10240;
BoxFitness benchmark = CostWeighting.JAVA_INTEGER;

BoxTemplate best = BoxTemplateHelper.getBestFor(
    benchmark, hoursOn,
    mbUpload, mbDownload);

// Deploy 5 of the boxes with an Ubuntu OS
BoxConfiguration os = BasicBoxConfig.getUbuntu(best);
ReservedBox res = provider.reserve(best, 5, os);

BoxInstance[] insts = provider.instantiate(res);

for (BoxInstance instance: insts) {
    String id = instance.getId();
    String provider = instance.getProvider().getName();
    String ip = instance.getPublicIp();

    System.out.printf('Instance %s on %s has IP %s\n',
                      id, provider, ip);

    // If they're EC2, print out the datacentre too
    if (instance instanceof EC2BoxInstance) {
        String zone = ((EC2BoxInstance)instance).getZone();
        System.out.printf(' In datacentre %s\n', zone);
    }
}
}

```

In the above example we set up a meta-provider, *Union Provider*, which knows about EC2, Flexiscale and NewServers—in effect we are setting those providers we have established accounts with. We call *find* with restrictions, the results from all three providers are combined – making the selection of the cheapest provider completely transparent to our code. The cheapest template is selected using, as our fitness function, an estimate of the cost of 48 hours of use with 1GB uploaded and 10GB downloaded. Once complete, a generic helper, *BasicBoxConfig*, is used to find an Ubuntu operating system in the providers list of operating system images. We then reserve and instantiate 5 boxes as in the previous example.

Our generic selection process means that as soon as the abstraction layer knows about new options (whether through the vendor API or in its own code), the user can take advantage of this with no changes to their code. In addition, if the users Flexiscale account is out of credit they can easily (and transparently) failover to their EC2 account. The fitness function uses a table of performance values (cached in the provider implementations) we have computed for each template for various instance types from providers; this allows us to simply

specify the *performance* we want rather than worry about the differences between various provider virtualisation platforms.

3 Use Case – A Media Service Cloud

The cloud provider library described here has been used in support of the PRISM media service cloud [9] to assist in managing a dynamic infrastructure of services for on-demand media provision of transcoding, storage and federated metadata indexing (Fig. 5). The PRISM infrastructure supports access to BBC content from a set-top box, web browser and from a range of media devices, such as mobile phones and games consoles. The infrastructure is currently being used for a large-scale user trial that provides access to all of the BBC's content.

The PRISM system uses resources within the Belfast e-Science Centre, British Telecom and BBC datacentres to provide core infrastructure. When user demand for content is high, PRISM services are hosted on utility resources from 3rd party resource providers. The infrastructure is managed in a uniform way as a collection of cloud providers—partner clouds that are the core infrastructure (BeSC, BBC and BT) and utility providers that can be called upon for extra resources. The PRISM application is written in a way similar to the examples outlined above—initially the baseline services are allocated from a cloud provider. When the services within PRISM detect that the system is overloaded, additional resources are deployed from a suitable provider. The cost model used in selection ensures that, initially, core infrastructure is selected; as this is used and no resources are available in the core clouds, 3rd party clouds are used to provide supporting services. This model has proven to be highly flexible—it has meant that the loss of a core infrastructure cloud was automatically compensated for with increased utility provider use.

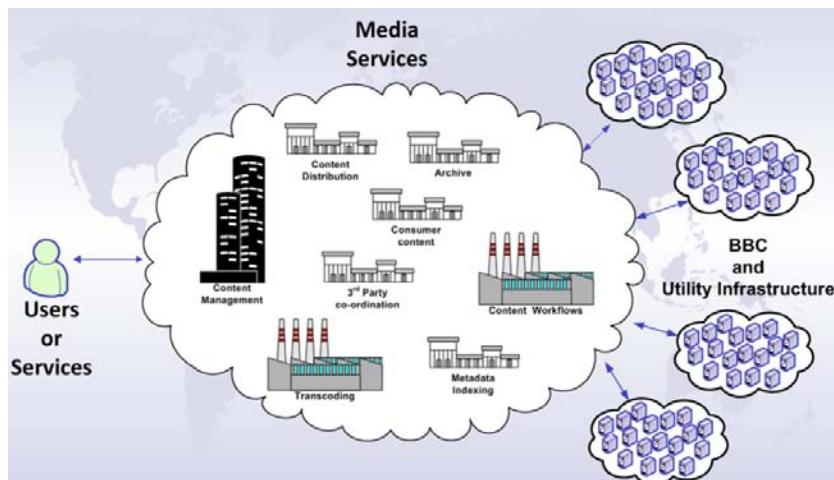


Fig. 5. PRISM Media Services Cloud

3.1 Evaluation of Use within PRISM

This abstraction layer, in combination with the supporting dynamic service deployment framework, has saved a large amount of time and development effort in PRISM, as well as dynamic infrastructure costs. The restrictions supplied put a heavy weight on the amount and cost of bandwidth to existing machines, allowing the most cost-effective locations to be chosen (internal infrastructure, where available). This has been demonstrated twice with the PRISM infrastructure:

1. Building contractors cut through our dedicated fibre optic link to the BBC Northern Ireland datacentre; the system dynamically failed over to free infrastructure in the BeSC datacentre
2. The SAN attached to the PRISM transcoding cloud failed; the framework failed over to EC2 for the 2 days it required to correct the problem, costing \$51.32 per day (67% of the cost was the transcoder VMs), however this came with a performance drop due to content being served over a 250mbit line

4 Conclusion and Future Work

As the cloud computing landscape gains acceptance in research and established businesses, users will be looking for cloud platform independence. This is crucial if cloud computing is to fulfil its promise of providing a robust infrastructure. Amazon is currently the major player, although its competitors are offering compelling feature sets for researchers and enterprises looking to deploy existing applications with minimal changes.

Our main aim was to simplify the allocation, management and discarding of on-demand resources from a multitude of providers. The initial generic API provides this functionality cleanly and efficiently and has back-ends for the APIs of Amazon EC2, Flexiscale, NewServers and the BeSC Service Hosting Cloud. It has been used directly for applications that want fine-grained control over their allocated VMs and by our service deployment framework that is available to UK NGS users. It has enabled our developers to be more experimental with the design of our own Service Hosting Cloud, as we must only maintain the binding to our abstraction layer to ensure our internal services can continue to use the service uninterrupted.

As Amazon and other vendors improve their feature sets we are seeing yet more common areas – in particular, customisable SAN mount, static IP address management, and resource resilience and scaling. The groundwork for the inclusion of these features has been laid, however they were not the main focus of its creation so they will be incorporated at a later date.

References

1. Wang, L., Jinjun Chen, W.J.: *Grid Computing: Infrastructure, Service, and Applications*. CRC Press, Boca Raton (2009)
2. Teragrid (2008), <http://www.teragrid.org>

3. Wladawsky-Berger, I.: Cloud computing, grids and the upcoming cambrian explosion in IT (2008)
4. Amazon, Inc.: Amazon Elastic Compute Cloud (2008),
<http://aws.amazon.com/ec2/>
5. Xcalibre, Inc. (2008), <http://www.flexiscale.com>
6. AppNexus, Inc. (2008), <http://www.appnexus.com>
7. NewServers, Inc. (2008), <http://www.newservers.com>
8. ElasticHosts, Ltd. (2008), <http://www.elastichosts.com>
9. Perrott, R., Harmer, T., Lewis, R.: e-Science infrastructure for digital media broadcasting. Computer 41, 67–72 (2008)
10. Harmer, T.: Gridcast—a next generation broadcast infrastructure? Cluster Computing 10, 277–285 (2007)
11. CRISP: Commercial R3 IEC Service Provision (2008), <http://crisp-project.org>
12. Eucalyptus Systems: The Eucalyptus Open-source Cloud-computing System (2008), <http://open.eucalyptus.com/>
13. SLA@SOI Project (2008), <http://sla-at-soi.eu>

MPI Applications on Grids: A Topology Aware Approach

Camille Coti¹, Thomas Herault^{1,2}, and Franck Cappello¹

¹ INRIA, F-91893 Orsay, France

coti@lri.fr, fci@lri.fr

² Univ Paris Sud, LRI, F-91405 Orsay, France

herault@lri.fr

Abstract. Porting on grids complex MPI applications involving collective communications requires significant program modification, usually dedicated to a single grid structure. The difficulty comes from the mismatch between programs organizations and grid structures: 1) large grids are hierarchical structures aggregating parallel machines through an interconnection network, decided at runtime and 2) the MPI standard does not currently provide any specific information for topology-aware applications, so almost all MPI applications have been developed following a non-hierarchical and non-flexible vision. In this paper, we propose a generic programming method and a modification of the MPI runtime environment to make MPI applications topology aware. In contrary to previous approaches, topology requirements for the application are given to the grid scheduling system, which exposes the compatible allocated topology to the application.

1 Introduction

Porting MPI applications on grids and getting acceptable performance is challenging. However two clear user motivations push researchers to propose solutions: 1) The *de-facto* standard for programming parallel machines is the Message Passing Interface (MPI). One of the advantages of MPI is that it provides a single, well defined programming paradigm, based on explicit message passing and collective communications. It is interesting to consider an MPI for grids, since complex applications may use non trivial communication schemes both inside and between clusters; 2) Because of their experience in parallel machines, many users wish to port their existing MPI applications, but redeveloping large portions of their codes to fit new paradigms requires strong efforts.

Not all parallel applications will perform well on a grid, and in general optimizations are required to reach acceptable performance. However, computation intensive applications following the master-worker or monte-carlo approaches are good candidates and some of them have been ported and executed successfully on grids [1] [2] [4].

In this paper, we investigate the issue of porting more complex MPI applications on grids. More specifically, we consider applications involving some collectives communications. In order to port complex MPI applications on grids, several issues have to be addressed. In [8], we already addressed the problem of designing an efficient MPI on grids and enabling transparent inter-cluster communications. However, with this

framework, MPI applications cannot take full advantage of the grid performance. Indeed, the communication pattern does not differentiate communications between nodes inside a cluster and remote nodes. As a consequence, the application may continuously communicate between clusters, with a significant impact on performances.

The difficulty of porting complex MPI applications on grids comes from 1) the difference between MPI programs organization and grid structures and 2) the static organization of existing MPI programs that does not fit with the diversity of grid structures. Cluster of clusters grids are intrinsically hierarchical structures where several parallel machines are connected through a long-distance interconnection network. In contrary, MPI standard does not currently provide any specific information on the topology, so almost all MPI applications have been developed following a non-hierarchical vision. In addition all grids differ in their topology and there is no mechanism in MPI to self-adapt the topology of the application to the one of the execution environment.

Previous works addressed the lack of topology awareness in MPI by exposing the topology of the available resources to the application. However, this approach requires a strong effort from the application to adapt itself to potentially any kind of resources that can be available at the time of submission, and is a key lock to building topology-aware MPI applications for grids. Such applications need to have a generic computation pattern that can adapt itself to any communication pattern, and such applications are very difficult (and sometimes impossible) to program. Our approach to address this problem is to combine: a) a modification of the MPI program organization to make it hierarchical and flexible b) a description by the programmer of its hierarchical communication pattern through a virtual topology and c) a mapping of the virtual topology to the physical one as provided by the grid reservation and scheduling service.

Typically in our approach, the application developer adapts the application code in a hierarchical approach and describes its "virtual" computation and communication patterns in a companion file. The application developer specifies in the companion file properties for specific processes and network requirements between nodes. To execute the application, the user submits it to the grid by providing as usual the binary, its parameters and data files, the number of desired nodes, and the companion file. We assume that the grid reservation and scheduling system assigns physical resources to the application according to a best effort matching with the user requirements and the application's companion file. This assumption corresponds to the architecture proposed in the QoSGrid project, and scheduling techniques to allocate resources corresponding to the developer and user requirements are described in [7, 16, 17]. The modified MPI system adapts the collective operations to optimize communications on the physical topology, and exposes the virtual topology required by the developer to the application, thus optimizing communication patterns to the hierarchical topology. Since communication costs can vary by orders of magnitude between two consecutive levels of topology hierarchy, performances can greatly benefit from collective operations that adapt their point-to-point communications pattern to the physical topology.

We present 4 main contributions to expose our approach in details and demonstrate its effectiveness: 1) the method to make MPI applications adapt to grids' hierarchy; 2) the presentation and performance evaluation of a grid-enabled MPI middleware, featuring topology awareness; 3) the evaluation of adapted collective operations that fit with

the topology of the grid using topology information, namely Broadcast, Reduce, Gather, Allgather and Barrier; 4) the description and evaluation of a grid-enabled application that takes advantage of our approach.

2 Related Work

A few approaches tried to tackle the topology adaptation problem (e.g. PACX-MPI and MPICH-G [10, 14]) by publishing a topology description to the application at runtime. The Globus Toolkit (GT) [9] is a set of software that aims to provide tools for an efficient use of grids. MPICH [11] has been extended to take advantage of these features [14] and make an intensive use of the available resources for MPI applications. MPICH-G2 introduced the concept of colors to describe the available topology. It is limited to at most four levels, that MPICH-G2 calls: WAN, LAN, system area and, if available, vendor MPI. Those four levels are usually enough to cover most use-cases. However, one can expect finer-grain topology information and more flexibility for large-scale grid systems. These approaches expose the *physical* topology for the application, which has to adapt by itself to the topology: this is the major difference with our approach. Practical experiments demonstrated that it is a difficult task to compute an efficient communication scheme without prior knowledge on the topology: the application must be written in a completely self-adaptive way.

Used along with Globus, Condor-G uses a technique called *gliding-in* [21] to run Condor jobs on a pool of nodes spanning several administrative domains. This way, a pool of Condor machines is made of the aggregation of those remote resources, the personal matchmaker and the user's Condor-G agent. This technique can be a solution for executing master-worker applications on a grid, but most non grid-specific applications are written in MPI and cannot be executed with Condor. Moreover, global operations like broadcasts and reductions cannot be done with Condor.

Collective operations have been studied widely and extensively in the last decades. However, as pointed out in [20] proposed strategies are optimal in homogeneous environments, and most often with a power-of-two number of processes. Their performance are drastically harmed in the heterogeneous, general case of number of nodes.

Topology-discovery features in Globus have been used to implement a topology-aware hierarchical broadcast algorithm in MPICH-G2 [13]. However, complex applications require a larger diversity of collective operations, including reductions, barrier, and sometimes all-to-all communications.

Grid-MPI [18] provides some optimized collective operations. The AllReduce algorithm is based on the works presented in [20]. The broadcast algorithm is based on [2]. Collective operations are optimized to make an intensive use of inter-cluster bandwidth, with the assumption that inter-cluster communications have access to a higher bandwidth than intra-cluster. However, 1) this is not always a valid assumption and 2) cluster of clusters grid have a large diversity of topology and results presented in [18] only concern 2 clusters.

Previous studies on hierarchical collective operations like [6] create a new algorithm for the whole operation. Our approach tries to make use of legacy algorithms whenever possible, i.e., in homogeneous sub-sets of the system (e.g., a cluster). MagPIe [15] is

an extension of MPICH for aggregations of clusters. MagPIe considers as a cluster any single parallel machine, which can be a network of workstations, SMPs or MPPs. It provides a set of collective operations based on a two-level hierarchy, using a flat tree for all the inter-cluster communications. This requirement strongly limits the scope of hardware configurations. Moreover, a flat tree may not always be the most efficient algorithm for upper-level communications.

3 Architecture

In this section we present how application developers can program their applications in order to make them fit to grids. We assume two kinds of topologies: the virtual topology, seen by the programmer, and the physical topology, returned by the scheduler. The virtual topology connects MPI processes or groups of processes in a hierarchical structure; the physical topology connects resources (core, CPU, cluster, MPP...) following a hierarchical structure. We consider that a physical topology is compatible with a virtual topology if the general structure of both topologies are matching and if the physical topology is not more scattered than the virtual topology (the physical topology preserves the geographical locality of inter-process communications).

For example, if the developer requested three groups for tightly coupled processes, the scheduler can map them on three clusters, or two clusters only: both physical topologies meet the requirements of the virtual topology, since the geographical locality is preserved.

We assume that the programmer designed the virtual topology without considering a specific physical topology. However, when the programmer developed an application, he can define requirements on the physical topology through parametrization of the virtual topology. Parameters in the virtual topology are link bandwidth, link latency, available memory... (these requirements are optional). The requirements are provided to the grid meta-scheduler, that tries to allocate nodes on a physical topology matching these requirements; the allocation algorithm and a topology aware Grid meta-scheduler are described in details in [27] and are not the object of this paper.

Besides, we have developed new collective operations in the MPI library that adapt themselves to the physical topology returned by the scheduler. If we consider the aforementioned example, global collective operations will be optimized for two subsets of processes instead of three subsets of processes as required by the virtual topology. The current version of the adaptation algorithm assumes that geographical locality always reduces the collective communication execution time.

Our collective operations use the best implementation of collective operations available for every computing resource in the hierarchical structure. Compared to collective operations for homogeneous environments discussed in Section 2, our collective operations adapt themselves to the physical topology.

To submit and execute a topology-aware application, the developer writes his application and describes processes or process groups and communications in a companion file called *jobProfile*. The *jobProfile* is submitted to the scheduler, that provides the list of allocated machines to the launcher. The application is deployed and started on this set of machines. The MPI runtime environment obtains the physical and virtual topologies

Vanilla Ray2mesh:	Hierarchical Ray2mesh:	
Broadcasts	Broadcasts	else /* worker */
if <i>I_am_master</i> :	if <i>I_am_central_master</i> :	upon receive chunk:
while(chunk)	while(chunk)	calculate ray tracing
distribute among workers	distribute among bosses	send results to the boss
receive results from workers	receive results from bosses	endif
else /* worker */	else	endif
upon receive chunk:	if <i>I_am_a_boss</i> :	Broadcast
calculate ray tracing	upon receive chunk:	AllToAll
send results to the master	while(chunk)	Output local result
endif	distribute among workers	
Broadcast	receive results from workers	
AllToAll	send results to the central mas-	
Output local result	ter or my upper-level boss	

Fig. 1. Ray2mesh, vanilla and hierarchical code versions

and transmits them to the MPI library (for collectives communications) and the application in order to identify the location of every MPI process in the virtual topology.

The following three subsections explain in more details how each step of the adaptation are done. Subsection 3.1 describes a specifically adapted application, Subsection 3.2 describes how the topology is adapted to the application’s requirements in terms of resources and communications, and Subsection 3.3 describes a set of collective operations designed to fit on the physical topology and knowledge about proximity between processes.

3.1 Grid-Enabled Application

The master-worker approach is used for a very wide range of parallel applications. Its major drawback is the single point of stress (master) creating a bottleneck. We consider the class of master-worker applications where parallel computations are done from one or several large partitionnable data sets, initially located on the central master. Partitions of the data set(s), that we call “chunks” are distributed to the workers during the execution, following a scheduling algorithm.

For these applications, after computing a chunk, a worker sends its result to the master and waits for a new chunk. Data prefetch could be used as an optimization to overlap communication and computation in order to reduce worker idle time [3]. However this approach requires strong modifications of the application, for both master and worker code and compromises the utilisation of external libraries in the application.

In a hierarchical communication and computation pattern, we introduce local masters in the virtual topology that can be used to relay data from the central master to the workers, and results from the workers to the central master. In the following, we call such a local master a *boss*. Bosses must be used at every intermediate level of the topology. A boss receives data from its upper-level boss, and sends it down to its lower-level boss or worker. Bosses are used in the virtual topology to reduce the number of cross-level communications and to foster locality of communications.

We have applied our hierarchical execution approach to Ray2mesh [12], a geophysics application that traces seismic rays along a given mesh containing a geographic area description. It uses the Snell-Descartes law in spherical geometry to propagate a wave

front from a source (earthquake epicenter) to a receiver (seismograph). In the following, we consider the master-worker implementation of Ray2mesh.

The execution of Ray2mesh can be split up into three phases (see Figure 1). The first one consists of successive collective operations to distribute information to the computation nodes. The second phase is the master-worker computation itself. The third phase is made of collective operations to give information from all workers to all others, before they can output their part of the final result.

We use the topological information to build a multi-level implementation of the three phases involved in Ray2mesh to make the communication pattern fit with the typically hierarchical topology of the grid.

This approach provides the same attractive properties as a traditional master-worker application, with any number of levels of hierarchy. Hence, it performs the same load-balancing, not only among the workers, but also among the bosses. This property allows suiting to different sizes of clusters and different computation speeds. Moreover, it allows each boss handling fewer data requests than in an organization with a unique master.

3.2 Hardware Resources and Application Matching

The communications of an application follow a certain pattern, which involve some requirements to be fulfilled by the physical topology. For example, tightly-coupled processes will require low-latency network links, whereas some processes that do not communicate often with each other but need to transfer large amounts of data will have bandwidth requirements. Those requirements can be described in a *JobProfile*. The jobProfile is submitted to the grid scheduler, that tries to allocate resources with respect to the requirements by mapping the requested virtual topology on available resources whose characteristics match as tightly as possible the ones requested in the JobProfile.

The JobProfile describes the process groups involved in the computation, in particular by specifying the number of processes in each group and requirements on inter- and intra-cluster communication performances. Some parameters are left blank, and filled by the meta-scheduler with the characteristics of the obtained mapping.

The *groupId* defined in the jobProfile will be passed to the application at runtime, along with the virtual topology of the resources that were allocated to the job. Using *groupIds*, it is simple to determine during the initialization of the application which group a given process belongs to, and which processes belong to a given group. The virtual topology description is passed like it was done for MPICH-G2 (cf Section 2), using an array of colors. Basically, two processes having the same color at a same hierarchy depth belong to the same group. In MPICH-G2, the depth of a hierarchy is limited to four. Our virtual topologies does not have this limitation.

3.3 Adapted Collective Operations

Collective operations are one of the major features of MPI. A study conducted at the Stuttgart High-Performance Computing Center [20] showed that on their Cray T3E, they represent 45% of the overall time spent in MPI routines.

To the best of our knowledge, no equivalent study was ever done on a production grid during such a long period. However, one can expect non-topology-aware

collective communications to be even more time-consuming (with respect to all the other operations) on an heterogeneous platform.

As in other Grid-MPI work (*cf* Section 2), our MPI for grids features collective communication patterns adapted to the physical topology in order to optimize them. In the following paragraphs, we describe which collective operations have been modified for topology-awareness and how they have been modified.

MPI_Bcast Sending a message between two clusters takes significantly more time than sending a message within a cluster. The latency for small synchronization messages, can be superior by several orders of magnitude, and the inter-cluster bandwidth is shared between all the nodes communicating between clusters.

The broadcast has been modified for exploiting the hierarchy of the physical topology. The root of the broadcast, if it belongs to the top-level master communicator, broadcasts the message along this top-level communicator. Otherwise, the root process sends the message to a top-level process which does exactly the same thing afterwards. Each process then broadcasts the message along its “sub-masters” communicator, until the lowest-level nodes are reached.

MPI_Reduce Using associativity of the operator in the Reduce operation, it can be made hierarchical as follows: each lowest level cluster performs a reduction towards their master, and for each level until the top level is reached the masters perform a reduction toward their level master.

MPI_Gather A *Gather* algorithm can also be done in a hierarchical way: a root is defined in each cluster and sub-cluster, and an optimized gather algorithm is used within the lowest level of hierarchy, then for each upper level until the root is reached.

The executions among sub-masters gather buffers which are actually aggregations of buffers. This aggregation minimizes the number of inter-cluster communications, for the cost of only one trip time while making a better use of the inter-cluster bandwidth.

MPI_Allgather aggregates data and makes the resulting buffer available on all the nodes. It can be done in a hierarchical fashion by successive *Allgather* operations from the bottom to the top of the hierarchy, followed by a hierarchical *Bcast* to propagate the resulting buffer. *MPI_Barrier* is similar to an *MPI_Allgather* without propagating any data.

4 Experimental Evaluation

We modified the runtime environment and the MPI library of the QosCosGrid Open MPI implementation presented in [8] to expose the virtual topology to the application. We also implemented the collective operations described in Section 3.3 using MPI functions.

We conducted the experiments on two traditional platforms of high performance computing: clusters of workstations with GigaEthernet network and computational grids. These experiments were done on the experimental Grid'5000 [5] platform or some of its components.

First, we measure the efficiency of topology-aware collective operations, using micro-benchmarks to isolate their performance. Then we measure the effects of hierarchy on a master-worker data distribution pattern and the effects on the Ray2mesh application. In the last section, we present a graph showing the respective contribution of the hierarchical programming and topology aware collective operations on the application performance.

4.1 Experimental Platform

Grid'5000 is a dedicated reconfigurable and controllable experimental platform featuring 13 clusters, each with 58 to 342 PCs, inter-connected through Renater (the French Educational and Research wide area Network). It gathers roughly 5,000 CPU cores featuring four architectures (Itanium, Xeon, G5 and Opteron) distributed into 13 clusters over 9 cities in France.

For the two families of measurement we conducted (cluster and grid), we used only homogeneous clusters with AMD Opteron 248 (2 GHz/1MB L2 cache) bi-processors. This includes 3 of the 13 clusters of Grid'5000: the 93-node cluster at Bordeaux, the 312-node cluster at Orsay, a 99-node cluster at Rennes. Nodes are interconnected by a Gigabit Ethernet switch.

We also used QCG, a cluster of 4 multi-core-based nodes with dual-core Intel Pentium D (2.8 GHz/2x1MB L2 cache) processors interconnected by a 100MB Ethernet network.

All the nodes were booted under linux 2.6.18.3 on Grid'5000 and 2.6.22 on the QCG cluster. The tests and benchmarks are compiled with GCC-4.0.3 (with flag -O3). All tests are run in dedicated mode.

Inter-cluster throughput on Grid'5000 is 136.08 Mb/s and latency is 7.8 ms, whereas intra-cluster throughput is 894.39 Mb/s and latency is 0.1 ms. On the QCG cluster, shared-memory communication have a throughput of 3979.46 Mb/s and a latency of 0.02 ms, whereas TCP communications have a throughput of 89.61 Mb/s and a latency of 0.1 ms.

4.2 Collective Operations

We ran collective operation benchmarks on 32 nodes across two clusters in Orsay and Rennes (figures 2a-b). A configuration with two clusters is an extreme situation to evaluate our collective communications: a small and constant number of inter-cluster messages are sent by topology-aware communications, whereas $O(\log(p))$ (where p is the total number of nodes) inter-cluster messages are sent by standard collective operations.

We also conducted some experiments on the QCG cluster with 8 processes mapped on each machine. Although this mapping oversubscribes the nodes (8 processes for 2 available slots), our benchmarks are not CPU-bound, and this configuration enhances the stress on the network interface. Measurements with a profiling tool validated the very low CPU usage during our benchmark runs.

We used the same measurement method as described in [14], using the barrier described in Section 3.3 to synchronize time measurements.

Since we implemented our hierarchical collective operations in MPI, some pre-treatment of the buffers may be useful. Messages are pre-cut and sent chunk after chunk. Then it is possible to pipeline the successive stages of the hierarchical operation. It appeared to be particularly useful when shared-memory communications were involved, allowing fair system bus sharing.

Figures 2(a) and 2(b) picture comparisons between standard and hierarchical MPI_Bcast and MPI_Reduce on Grid'5000. Message pre-cutting appeared to be useful for MPI_Bcast, whereas it was useless for MPI_Reduce, since big messages are already split by the algorithm implemented in Open MPI.

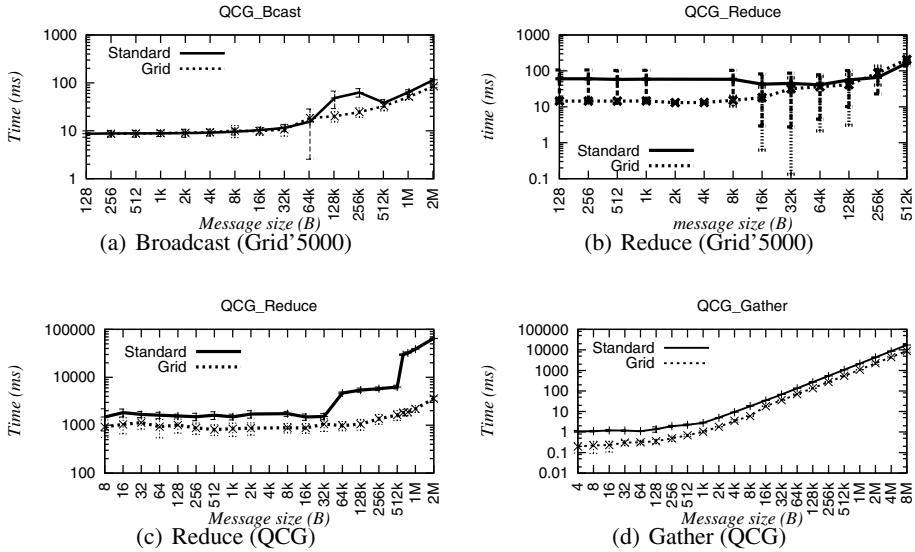


Fig. 2. Comparison between standard and grid-enabled collective operations on a grid

One can see that, as expected, hierarchical MPI_Bcast (Figure 2(a)) always performs better than the standard implementation. Moreover, pre-cutting and pipelining permits to avoid the performance step around the eager/rendez-vous mode transition.

When messages are large regarding communicator size, MPI_Reduce (Figure 2(b)) in Open MPI is implemented using a pipeline mechanism. This mechanism allows communication costs to be dominated by the high throughput of the pipeline rather than the latency of a multi-steps tree-like structure. Hierarchy shortens the pipeline: then its latency (*i.e.*, time to load the pipeline) is smaller and it performs better on short messages. But for large messages (beyond 100 kB), the higher throughput of a longer pipeline outperforms the latency-reduction strategy. In this case, hierarchical communications are not an appropriate approach, and a single flat pipeline performs better.

Figures 2(c) and 2(d) picture comparisons between standard and hierarchical MPI_Reduce and MPI_Gather on the QCG cluster. On a cluster of multi-cores, collective operations over shared-memory outperform inter-machine TCP communications significantly enough to have a negligible cost. Therefore, on a configuration including a smaller number of physical nodes, inducing more shared-memory communications, our hierarchical MPI_Reduce performs better (Figure 2(c)).

4.3 Adapted Application

The execution phases of Ray2mesh are presented in Section 3.1. It is made of 3 phases: two collective communication phases and a master-worker computation phase in between them. When the number of processes increases, one can expect the second phase to be faster but the first and third phases to take more time, since more nodes are involved in the collective communications.

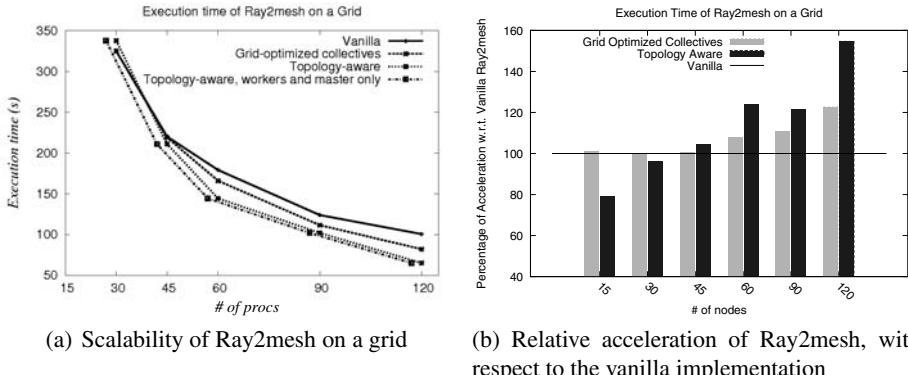


Fig. 3. Comparison of vanilla Ray2mesh with vanilla Ray2mesh using optimized collective communications, and fully topology-aware Ray2mesh

Figure 3(a) presents the scalability of Ray2mesh under three configurations: standard (vanilla), using grid-adapted collective operations, and using a hierarchical master-worker pattern and grid-adapted collective operations. Those three configurations represent the three levels of adaptation of applications to the Grid. The standard deviation is lower than 1% for each point. The fourth line represents the values of the last configuration, measured with the same number of computing elements as in the first configuration, thus removing the local boss in the process count.

First of all, Ray2mesh scales remarkably well, even when some processes are located on a remote cluster. When a large number of nodes are involved in the computation, collective operations represent an important part of the overall execution time. We can see the improvement obtained from grid-enabled collectives on the “grid-optimized collectives” line in Figure 3(a). The performance gain for 180 processes is 9.5%.

Small-scale measurements show that the grid-enabled version of Ray2mesh does not perform as well as the standard version. The reason is that several processes are used to distribute the data (the bosses) instead of only one. For example, with 16 processes distributed on three clusters, 15 processes will actually work for the computation in a single-master master-worker application, whereas only 12 of them will contribute to the computation on a multi-level (two-level) master-worker application. A dynamic adaptation of the topology according to the number of involved node would select the “non hierarchical” version for small numbers of nodes and would select the hierarchical version when the number of nodes exceeds 30.

However, we ran processes on each of the available processors, regardless of their role in the system. Bosses are mainly used for communications, whereas workers do not communicate a lot (during the master-worker phase, they communicate with their boss only). Therefore, a worker process can be run on the same slot as a boss without competing for the same resources. For a given number of workers, as represented by the “workers and master only” line in Figure 3(a), the three implementations show the same performance for a small number of processes, and the grid-enabled implementations are more scalable. The performance gain for 180 processes is 35% by adding only 3 dedicated nodes working exclusively as bosses.

The relative acceleration with respect to the vanilla implementation is represented Figure 3(b). We can see that the application speed is never harmed by optimized collective operations and performs better on large scale, and a topology-aware application is necessary to get a better speedup for large-scale application.

5 Conclusion

In this paper, we proposed a new topology-aware approach to port complex MPI applications on grid through a methodology to use MPI programming techniques on grids. First we described a method to adapt master-worker patterns to grids' hierarchical topology. We used this method to implement a grid-enabled version of the Ray2mesh geophysics applications featuring a multi-level master-worker pattern and our hierarchical collective operations. Then we proposed a way to describe the communication patterns implemented in the application in order to match the application's requirements with the allocated physical topology. In the last part we presented a set of efficient collective operations that organize their communications with respect to the physical topology in order to minimize the number of high-latency communications.

Experiments showed the benefits of each part of this approach and their limitations. In particular, experiments showed that using optimized collectives fitted to the physical topology of the grid induce a performance improvement. They also showed that adapting the application itself can improve the performances even further.

We presented an extension of the runtime environment of an MPI implementation targeting institutional grids to provide topology information to the application. These features have been implemented in an MPI library for grids.

Acknowledgements. Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, an initiative from the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners (see <https://www.grid5000.fr>), and founded by the QosCosGrid European project (grant number: FP6-2005-IST-5 033883).

References

- [1] Atanassov, E.I., Gurov, T.V., Karaivanova, A., Nedjalkov, M.: Monte carlo grid application for electron transport. In: Alexandrov, V.N., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2006. LNCS, vol. 3993, pp. 616–623. Springer, Heidelberg (2006)
- [2] Barnett, M., Gupta, S., Payne, D.G., Shuler, L., van de Geijn, R., Watts, J.: Building a high-performance collective communication library. In: Proc. of SC 1994, pp. 107–116. IEEE, Los Alamitos (1994)
- [3] Boutammine, S., Millot, D., Parrot, C.: An adaptive scheduling method for grid computing. In: Nagel, W.E., Walter, W.V., Lehner, W. (eds.) Euro-Par 2006. LNCS, vol. 4128, pp. 188–197. Springer, Heidelberg (2006)
- [4] Branford, S., Sahin, C., Thandavan, A., Weihrauch, C., Alexandrov, V.N., Dimov, I.T.: Monte carlo methods for matrix computations on the grid. Future Gener. Comput. Syst. 24(6), 605–612 (2008)

- [5] Cappello, F., Caron, E., Dayde, M., et al.: Grid'5000: A large scale and highly reconfigurable grid experimental testbed. In: Proc. The 6th Intl. Workshop on Grid Computing, pp. 99–106 (2005)
- [6] Cappello, F., Fraigniaud, P., Mans, B., Rosenberg, A.L.: HiCoHP: Toward a realistic communication model for hierarchical hyperclusters of heterogeneous processors. In: Proc. of IPDPS. IEEE, Los Alamitos (2001)
- [7] Charlot, M., De Fabritis, G., Garcia de Lomana, A.L., Gomez-Garrido, A., Groen, D., et al.: The QosCosGrid project. In: Ibergrid 2007 conference, Centro de Supercomputacion de Galicia (2007)
- [8] Coti, C., Herault, T., Peyronnet, S., Rezmerita, A., Cappello, F.: Grid services for MPI. In: Proc. CCGRID, pp. 417–424. IEEE, Los Alamitos (2008)
- [9] Foster, I.T.: Globus toolkit version 4: Software for service-oriented systems. *J. Comput. Sci. Technol.* 21(4), 513–520 (2006)
- [10] Gabriel, E., Resch, M.M., Beisel, T., Keller, R.: Distributed computing in a heterogeneous computing environment. In: Alexandrov, V.N., Dongarra, J. (eds.) PVM/MPI 1998. LNCS, vol. 1497, pp. 180–187. Springer, Heidelberg (1998)
- [11] Gropp, W., Lusk, E., Doss, N., Skjellum, A.: High-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing* 22(6), 789–828 (1996)
- [12] Grunberg, M., Genaud, S., Mongenet, C.: Parallel seismic ray tracing in a global earth model. In: Proc. of PDPTA, vol. 3, pp. 1151–1157. CSREA Press (2002)
- [13] Karonis, N.T., de Supinski, B.R., Foster, I., Gropp, W., Lusk, E., Bresnahan, J.: Exploiting hierarchy in parallel computer networks to optimize collective operation performance. In: Proc. of SPDP, pp. 377–386. IEEE, Los Alamitos (2000)
- [14] Karonis, N.T., Toonen, B.R., Foster, I.T.: MPICH-G2: A grid-enabled implementation of the message passing interface. In: CoRR, cs.DC/0206040 (2002)
- [15] Kielmann, T., Hofman, R.F.H., Bal, H.E., Plaat, A., Bhoedjang, R.A.F.: MAGPIE: MPI's collective communication operations for clustered wide area systems. In: Proc. of PPoPP. ACM Sigplan, vol. 34.8, pp. 131–140. ACM Press, New York (1999)
- [16] Kravtsov, V., Carmeli, D., Schuster, A., Yoshpa, B., Silberstein, M., Dubitzky, W.: Quasi-opportunistic supercomputing in grids, hot topic paper. In: Proc. of HPDC (2007)
- [17] Kravtsov, V., Swain, M., Dubin, U., Dubitzky, W., Schuster, A.: A fast and efficient algorithm for topology-aware coallocation. In: Bubak, M., van Albada, G.D., Dongarra, J., Sloot, P.M.A. (eds.) ICCS 2008, Part I. LNCS, vol. 5101, pp. 274–283. Springer, Heidelberg (2008)
- [18] Matsuda, M., Kudoh, T., Kodama, Y., Takano, R., Ishikawa, Y.: TCP adaptation for MPI on long-and-fat networks. In: Proc. of CLUSTER, pp. 1–10. IEEE, Los Alamitos (2005)
- [19] Nascimento, P., Sena, C., da Silva, J., Vianna, D., Boeres, C., Rebello, V.: Managing the execution of large scale mpi applications on computational grids. In: Proc. of SBAC-PAD, pp. 69–76 (2005)
- [20] Rabenseifner, R.: Optimization of collective reduction operations. In: Bubak, M., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2004. LNCS, vol. 3036, pp. 1–9. Springer, Heidelberg (2004)
- [21] Thain, D., Tannenbaum, T., Livny, M.: Condor and the grid. In: Grid Computing: Making the Global Infrastructure a Reality. John Wiley & Sons Inc., Chichester (2002)

Topic 7

Peer-to-Peer Computing

Introduction

Ben Zhao*, Paweł Garbacki*, Christos Gkantsidis*, Adriana Iamnitchi*, and Spyros Voulgaris*

After a decade of intensive investigation, peer-to-peer computing has established itself as an accepted research field in the general area of distributed systems. Peer-to-peer computing can be seen as the democratization of computing overthrowing traditional hierarchical designs favored in client-server systems largely brought about by last-mile network improvements which have made individual PCs first-class citizens in the network community. Much of the early focus in peer-to-peer systems was on best-effort file sharing applications. In recent years, however, research has focused on peer-to-peer systems that provide operational properties and functionality similar to those shown by more traditional distributed systems. These properties include stronger consistency, reliability, and security guarantees suitable to supporting traditional applications such as databases.

Twenty two papers were submitted to this years Peer-to-Peer Computing track, out of which only eight were accepted. Three of these propose optimizations to the overlay network construction, maintenance and analysis; two papers address the problem of peer sampling. One paper examines the issue of search in structured networks, another deals with reputation models, while the last explores challenges of IP-TV platforms based on peer-to-peer technology.

In “A least resistance path in reasoning about unstructured overlay networks”, parallels between unstructured peer-to-peer networks that work with random walks and electric circuits are drawn. The analogy between both concepts is proposed and evaluated experimentally on synthetic peer-to-peer networks metrics related to equivalent electric circuits. A unifying framework derived from tools developed for evaluation of resistor networks is shown to facilitate the analysis of overlay properties and the reasoning about algorithms for peer-to-peer networks.

In “Selfish Neighbor Selection in Peer-to-Peer Backup and Storage Applications”, a peer-to-peer storage and backup service in which peers choose their storage partners using an incentivized resource-driven model is studied. The study proposes an incentive scheme for peer cooperation that does not depend on external mechanisms. A model for peer resource availability is studied and it is shown that peers reach an equilibrium when they self organize into groups with similar availability profiles.

In “Zero-Day Reconciliation of BitTorrent Users With Their ISPs”, a modification of BitTorrent tracker reducing both bandwidth costs and download times is suggested. The improvement is achieved by instructing the tracker to return a subset of peers that are closest (in terms of latency) to the requesting peer, consequently improving the network locality of data exchange between peers

* Topic Chairs.

and decreasing the chance of traffic crossing ISP boundaries. The risk of overlay network partitioning is alleviated by adding a few randomly selected peers to the tracker response.

In “Uniform Sampling for Directed P2P Networks”, the problem of uniform random sampling in a peer-to-peer network is addressed with an approach based on a random walk. The transition probabilities in the overlay network graph are adjusted iteratively to guarantee that the transition matrix is doubly stochastic. The properties of the transition matrix guarantee that a random walk of a sufficient length lands in a uniformly random node.

In “Adaptive Peer Sampling with Newscast”, a number of improvements to the original peer sampling service built on top of Newscast protocol is proposed. Those improvements address limitations of Newscast which result in non-uniform sampling of peers as a consequence of heterogeneity in packet drop rates and peer probing intervals. Both limitations are resolved with a simple extension of the protocol and an adaptive self-control mechanism for its parameters.

In “SiMPSON: Efficient Similarity Search in Metric Spaces over P2P Structured Overlay Networks”, a system supporting similarity search in metric spaces is introduced. Each peer in this system first clusters its own data and then maps resulting data clusters to one-dimensional values which are indexed and stored in a structured overlay. Algorithms supporting advanced queries retrieving data from the overlay are components of the scheme.

In “Exploring the Feasibility of Reputation Models for Improving P2P Routing under Churn”, a churn-resilient reputation scheme for a dynamic peer-to-peer network is described. A proposed analytical model derives the reliability of the reputation scores based on transactions rate and peer uptime. Building on the conclusions from the analysis, a novel routing protocol for structured peer-to-peer systems is proposed. This protocol uses reputation scores to select the most reliable peer as the next hop on the routing path.

In “Surfing Peer-to-Peer IPTV: Distributed Channel Switching”, the reduction of the channel switching overhead in an IPTV system is studied by means of a newly proposed algorithm, the performance of which is assessed with simulations. The main idea of this algorithm is based on the observation that a list of peers actively involved in video distribution can be obtained more reliably from a the peer-to-peer network than from a central server which view of the network may be obsolete.

Finally, we would like to thank all authors and all reviewers of the papers submitted to this track for their work.

A Least-Resistance Path in Reasoning about Unstructured Overlay Networks*

Giorgos Georgiadis and Marina Papatriantafilou

Department of Computer Science and Engineering, Chalmers University of Technology, S-412
96 Göteborg, Sweden
Fax: +46-31-7723663
{georgiog,ptrianta}@chalmers.se

Abstract. Unstructured overlay networks for peer-to-peer applications combined with stochastic algorithms for clustering and resource location are attractive due to low-maintenance costs and inherent fault-tolerance and self-organizing properties. Moreover, there is a relatively large volume of experimental evidence that these methods are efficiency-wise a good alternative to structured methods, which require more sophisticated algorithms for maintenance and fault tolerance. However, currently there is a very limited selection of appropriate tools to use in systematically evaluating performance and other properties of such non-trivial methods.

Based on a well-known association between random walks and resistor networks, and building on a recently pointed-out connection with peer-to-peer networks, we tie-in a set of diverse techniques and metrics of both realms in a unifying framework. Furthermore, we present a basic set of tools to facilitate the analysis of overlay properties and the reasoning about algorithms for peer-to-peer networks. One of the key features of this framework is that it enables us to measure and contrast the local and global impact of algorithmic decisions in peer-to-peer networks. We provide example experimental studies that furthermore demonstrate its capabilities in the overlay network context.

1 Introduction

A commonly used classification of peer-to-peer resource-sharing networks distinguishes them in *structured* and *unstructured*, based on their approach to the resource-lookup problem [1] and whether they use a distributed data structure to navigate the network. An unstructured approach like flooding is easy to implement but can also be the cause of quite challenging problems such as high bandwidth load. However, several research results [2][3][4][5][6] indicate that a promising way to fix such shortcomings is through statistical methods such as the *random walker* technique, where the query messages are considered to perform random walks on the overlay. An added bonus of these techniques are the inherent, “built-in” *fault-tolerance* properties they exhibit, being resilient to failure of links or nodes.

Recently many authors pointed out the usefulness of non-trivial random walkers [7][8][9] in construction and querying of *interest-based networks*, where nodes sharing the same interests are closer together than others. However, these studies currently lack

* Research supported by the Swedish Research Council, contract number 60465601.

a unifying framework for the comparative study of the various random walks on different overlay networks. It is also the case that many significant results for non-trivial random walks come in context with physics-related phenomena (cf. [10] and references therein), and therefore cannot be applied directly in an overlay network context. However, there is a strong connection between random walks and electric circuit theory, especially *resistor networks* [11]. By focusing on unstructured overlays that use random walks to perform queries¹, we can translate them in terms of elementary electric circuit terms and then use established tools to do the analysis in a simplified manner. Bui and Sohier in [12][13] have made significant first steps in exploring this connection to resistor networks, along with other authors, albeit for simpler walks (e.g. [14] and references therein).

In this work we aim at connecting, refining and adapting distinct techniques that involve generalized random walks and electric networks in a peer-to-peer network context, as well as providing a methodology to help address, in a way both analytical and systematic, questions such as: What is the relation between random walks and the underlying overlay? Which random walks are better suited for which networks? How can we improve a random walk and which metrics can we use to evaluate its performance? Which applications are favored by various random walks and in which context? How can we model known and recurring problems of peer-to-peer networks using the electric network paradigm?

We expand previous work on the connection among random walks and electric circuits and we propose a framework to facilitate the analysis of random walk properties in the context of topologies. Central in this framework is the notion of a biased random walk upon locally stored information, which enables the use of relevant measures of interest for network navigation. We call this framework REPO: resistor-network-based analysis of peer-to-peer overlays. Using REPO it is possible to study a broad set of topics such as content replication efficiency, overlay construction for interest-based networks, load balancing, the effect of failures, as well as local policies, bounded resource handling issues and more.

Along with our framework, we present a basic set of tools that demonstrate its intuitiveness and we complement these tools with an illustrative study that addresses several of the key questions mentioned above. In particular, we (i) identify scenarios that differentiate a random walker's behavior in different peer-to-peer overlays, (ii) reason about options for improvement of the random walker's query efficiency and the underlying network's robustness, (iii) connect local policies to global phenomena observed in peer-to-peer overlays using resistor networks, (iv) address recurring problems in peer-to-peer networks using electric theory metrics, (v) suggest novel, analytic ways of employing established techniques in a peer-to-peer context. Let us note that the framework is expressed in a way that enables it to be used in various contexts regarding the overlay network and statistical methods, and can also address construction issues e.g. of interest-based networks [8].

Other Related Work. In [2], Adamic et al present an algorithm that performs queries with a random walker on an overlay following a power-law degree distribution,

¹ In the rest of the text the terms "query" and "random walker" are used interchangeably.

directing it at every step to the highest degree neighbor. Using mean field analysis they estimate the mean number of steps needed for the query but the fact that they use deterministic random walks prohibits them from using randomized techniques to speed up the lookup process, also causing obvious overloading problems to the highest degree nodes. Lv et al [15] perform non-deterministic, simple random walks using multiple walkers and by applying content replication along the lookup path they aim at speeding up consecutive lookups but they focus only on extensive simulations. In [3], Ata et al study experimentally a biased random walk similar to what we use to illustrate REPO, along with a version that directs the walker probabilistically towards the lowest connected nodes, and find cases where both of them are useful. Gkantsidis et al [4] use simple random walks both for querying and overlay construction, and show that they can decrease significantly the cost of flooding in many practical cases of unstructured overlay networks. Fraigniaud et al in [8] are the first, to the extent of our knowledge, to use a non-trivial random walk to build an overlay network. By using a deterministic random walk that makes transitions based on the node's degree in order to locate information, they create links between the issuer of the query and the answering node, resulting in an *interest network* where nodes are connected based on their interests. This deterministic random walk (see also [2]) may have some overloading problems, as mentioned by the authors; when used in interest-based overlay construction, it creates a network with heavy tail degree distribution similar to a power-law network and the deterministic walkers can overload the (relatively) few high degree nodes. However, the experimental study in [8] shows that each lookup succeeds in roughly logarithmic number of steps over the network size, indicating that a biased random walk can be quite efficient when used as a search strategy.

Structure of the Paper. Theoretical background concepts are presented in Section 2, including elements of graph theory, electric circuit theory and linear algebra needed in our analysis. The REPO framework is presented in Section 3, along with a basic set of tools and examples of theoretical analysis using the ideas therein. Section 4 presents a set of experimental studies based on the REPO framework, concerning a key set of topics of interest on peer-to-peer networks, such as content replication efficiency, tolerance to failures and effect of resource limitations. The final section presents conclusions and directions of future work.

2 Theoretical Background

Graph-Related Concepts. Let the overlay network be an undirected, weighted graph $G(V, E)$, $|V| = n$, $|E| = m$, with weights w_{ij} for every edge $e = (i, j)$. We will write Γ_i to denote the set of immediate neighbors of node i . Each node i has degree $k_i = |\Gamma_i|$ and *weighted degree* $d_i = \sum_{j \in \Gamma_i} w_{ji}$ the sum of weights over all edges adjacent to that node. The *connected components* of G are its maximal subgraphs, where a path exists between every pair of nodes in the same connected component.

The *adjacency matrix* A of a weighted graph has as elements $a_{ij} = w_{ij}$, and $a_{ij} = 0$ if there is no edge between i and j . The *Laplacian matrix* L of a weighted graph is the matrix $L = D - A$, where A is the adjacency matrix and D a diagonal matrix with

Table 1. Connections of peer-to-peer application notions and random walk concepts with resistor network terms

Peer-to-peer application	Random walk concept	Resistor network term	Description
Navigation	Edge weight	Conductance	The walker uses edge weights at every node to make a preferential choice for its next transition.
	Mean number of crossings	Current	When a unit current is injected into node a and exits at node b , the current at every edge e is the mean number of crossings of e by the walker at the direction of the current, when he starts at node a and before he reaches b [20].
Query efficiency	Return probability	Potential	When a unit voltage is applied between nodes a and b ($v_a = 1, v_b = 0$), the potential at every node is the probability for the walker to reach node a before reaching node b , when starting from that node [20].
	Mean commute time	Effective resistance	The effective resistance between any two points on the resistor network is proportional to the commute time of the walker between the same points [21].
	Mean first-passage time	None	The mean time needed for the walker to reach a destination node for the first time, for a fixed source-destination pair. Although there is no metric directly associated to the mean first-passage time, we can compute it using the Laplacian matrix of the graph.
Robustness	Node/edge connectivity	Algebraic connectivity	The algebraic connectivity is a lower bound for both node and edge connectivity of a graph [18].

values $d_{ii} = \sum_{k=1}^n w_{ik}$ [16]. We are mostly interested in the inverse of the Laplacian matrix, and although as singular matrix it has no usual inverse, we can use its Moore-Penrose generalized inverse (more on these matrices can be found in [16]). The second smallest eigenvalue of L is called *algebraic connectivity* and is a measure of how well connected and robust to failures is the corresponding graph. Some of its most well known properties [17] are that (i) the graph is disconnected if and only if its algebraic connectivity is zero and that (ii) the multiplicity of zero among the eigenvalues of the graph equals the number of connected components. Additionally, many simple operations that can be applied on a network have well documented effects on the algebraic connectivity [18]. For example, it is known that the formation of hypernodes leads to networks with higher algebraic connectivity [19].

Random Walks and Resistor Networks. We consider a discrete time random walk on the weighted graph with transition probability $p_{ij} = \frac{w_{ij}}{\sum_{k \in \Gamma_i} w_{ik}}$ from node i to j following edge (i, j) . The *mean first-passage time* $m(i, j)$ of node j is the mean number of steps a walker needs to reach node j for the first time, starting at i . Another metric of interest is the sum of $m(i, j)$ and $m(j, i)$, called *mean commute time* $c(i, j)$ between i and j .

For the definitions of elementary terms of electric circuit theory such as voltage, current, resistance and conductance, we point the reader at related books (e.g. [22]).

We are mostly interested in the *effective resistance* R_{xy} between two points x and y in a circuit, which is defined as the resistor we must use if we want to replace the entire circuit between these two points, in order to get the same current for the same difference in potential between these two points. The *effective conductance* C_{xy} is defined as the inverse of the effective resistance.

There is an important correspondence between graphs and electric circuits: a *resistor network* [11] is defined as a weighted undirected graph which corresponds to an electric circuit with the following characteristics: every graph node is treated as a checkpoint in a circuit, useful for measurements of potential, and every graph edge as a resistor with conductance equal to the edge's weight. A resistor bridges two checkpoints, as usual in electric circuits, if and only if there is an edge between the corresponding nodes. A simple example of this correspondence can be found in figure 1.

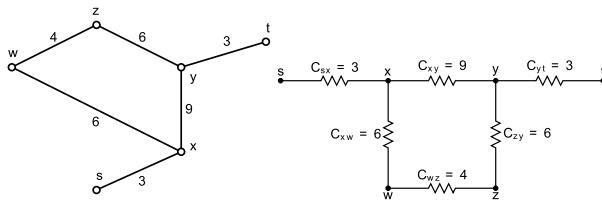


Fig. 1. A simple weighted graph and its corresponding resistor network

3 Framework Details and Essential Tools

Simple random walks have been used to route query messages in unstructured networks, in an effort to reduce the high communication costs of flooding [415]. However, simple random walks disregard completely the network's topology and link characteristics, e.g. the low latency of some high speed links to select destinations or the available bandwidth of the neighboring nodes. To address this handicap some authors use *biased* random walks that employ a locally stored metric to guide the random walker [208]. *Connectivity, bandwidth availability, latency times* or *socially-oriented information* are just some of the metrics that have been used and can be employed also in our framework. All these metrics are essentially information kept by nodes about their neighbors or statistics kept by each node and can be transmitted between them. These biased random walks have proven successful but their analysis lacks a unifying framework that will enable them to be studied comparatively in a specific overlay network or to study their performance in different networks.

The locally stored information at node i can be represented with a scalar metric m_i . This metric can be easily taken into account for the random walker's transition decisions by defining the transition probabilities as $p_{ij} = \frac{m_j}{\sum_{k \in \Gamma_i} m_k}$, where m_j is a metric stored in

node i related to its neighbor j , and m_k are similar metrics of all its neighbors $k \in \Gamma_i$. On the other hand, it is known (cf. previous section) that there is a correspondence between weighted graphs and electric circuits. Can we use these probabilities to define a resistor network? That will enable the use of established tools and intuitive nature of that

field. The lemma below shows that this holds for the generalized case of biased random walks; the interface and options for the study of problems in peer-to-peer applications that are enabled are given in more detail in the remaining parts of this section.

Lemma 1 (Equivalence lemma). *Suppose that the random walker makes a transition from node i to node j preferentially with probability $p_{ij} = \frac{m_j}{\sum_{k \in \Gamma_i} m_k}$, based on a locally stored metric m_j about node j , and stores the same information for all neighboring nodes $k \in \Gamma_i$. This biased random walk corresponds to a random walk on a weighted graph with weights $w_{ij} = m_i \cdot m_j$ for all edges (i, j) .*

Proof. If p_{ij} and p'_{ij} are the transitional probabilities of the biased random walk and the random walk on the weighted graph respectively, we get the following equation, implying that the two random walks are identical:

$$p'_{ij} = \frac{w_{ij}}{\sum_{k \in \Gamma_i} w_{ik}} = \frac{m_i \cdot m_j}{\sum_{k \in \Gamma_i} m_i \cdot m_k} = \frac{m_j}{\sum_{k \in \Gamma_i} m_k} = p_{ij} \quad (1)$$

Using the previous lemma, we represent any biased random walk which uses local information as a resistor network and this enables us to form a basis upon which different random walks on different networks can be compared and analyzed. We can then directly transfer some known connections of electric and random walk concepts to the biased walks we are modeling (summarized in table I).

3.1 Framework Interface

Laplacian Matrix. The Laplacian matrix of a graph plays a central part in proving analytical results on resistor networks, as well as evaluating experimentally key properties such as the mean first-passage and commute times. The most common usage of the Laplacian is through its inverse form as it is shown in the following paragraphs.

Effective Resistance. Another metric of interest is the effective resistance between any two nodes. On a local level, consider a node connected to his neighbors by regular resistances. The central node has a distance of one hop from each neighbor but since the random walker makes transitions based on the local resistances with different probabilities, some neighbors effectively come "closer" and some are pushed "further" away. On the other hand, Klein et al showed [11] that the effective resistance R_{xy} between two points is a *distance metric* on the corresponding graph, since it is non-negative, reflexive, symmetric and the triangle inequality holds. These two facts allow us to use the effective resistance to express the distance covered by a random walker. A recently proposed method by Bui et al [12] allows the computing of effective resistances between all node pairs with one matrix inversion, further allowing the computation of other metrics such as the mean first-passage times for all node pairs. Subsequently in this paper we will make use of the effective resistance in our experimental studies as well as illustrate its usage in an analytical setting.

Mean First-Passage and Commute Times. As Chandra et al showed [21], the effective resistance between two points expresses the mean commute time of the points in

the weighted graph, multiplied by a network-dependent constant. This result further reinforces the position of the effective resistance as a key metric in our framework. As we will see shortly, it is also connected to the mean first-passage time and can therefore be used to measure the expected query answer times and other metrics related to query efficiency (see also table 1). However, computing the effective resistance between two nodes is no trivial task. Klein et al [11] showed that computing the effective resistance is possible using linear algebra calculations over the Laplacian matrix. The following lemma simplifies the notation used by Klein et al in their solution and aims to illustrate the usage of the Laplacian matrix in analytical computation of large scale properties of the resistor network. Note that to use this method both nodes must belong to the same connected component.

Lemma 2. *The effective resistance R_{st} between nodes s and t in a resistor network of size n , where nodes s and t belong to the same connected component, is given by $R_{st} = c^T \cdot L^+ \cdot c$, where L^+ is the Moore-Penrose generalized inverse of the Laplacian matrix of the network and c is the external current vector of size n with elements $c(x) = 1$, if $x = s$, $c(x) = -1$, if $x = t$ and $c(x) = 0$ otherwise.*

Proof. To compute the effective resistance R_{st} between nodes s and t we inject one unit of current through node s and take one unit of current from node t . We know that there is a path from node s to node t for this unit current to follow since they belong to the same connected component, and the external current vector c of size n is the one specified above. Since we injected one unit of current, the effective resistance is the difference in potential between node s and t . In order to find the potential vector v we have to solve the linear system $L \cdot v = c \Rightarrow v = L^+ \cdot c$, where L^+ denotes the Moore-Penrose generalized inverse of the Laplacian matrix [16]. We multiply with c^T to get the difference in potential between s and t , and so $R_{st} = c^T \cdot v = c^T \cdot L^+ \cdot c$.

The above result is complemented by the following formula of Fouss et al in [23] that enables us to use the same matrix L^+ to compute the *mean first-passage time* of a walker, where l_{ij}^+ are elements of L^+ and d_{kk} are elements of D :

$$m(i, j) = \sum_{k=1}^n (l_{ik}^+ - l_{ij}^+ - l_{jk}^+ + l_{jj}^+) \cdot d_{kk} \quad (2)$$

Algebraic Connectivity. The algebraic connectivity is a tool in spectral analysis techniques, widely used in applications on large networks such as the Web graph (i.e. the PageRank algorithm [24]). However, only recently have these techniques come under the spotlight of the scientific community regarding their potential towards rigorous analysis of connectivity and fault tolerance properties of networks [25]. Algebraic connectivity in particular can be used in a wide variety of applications, ranging from network health indicator and an early warning system in security studies to an analytical tool useful to compare the robustness achieved by different peer-to-peer configurations.

Hypernode Formation. Hypernodes have been used in previous resistor network studies [13] but play a more central part in a peer-to-peer context. Their strength comes from the observation that from a random walker's viewpoint there is no need to distinguish between different nodes if they all share a common property that places them to

the endpoint of the walker's path. So, if some nodes are considered undistinguishable destinations for the random walker we can group them together in a hypernode, having the same set of neighbors as the entire collection of nodes, and study this walk as if they were inseparable. Note that the equivalent of a hypernode in a resistor network is short-circuiting nodes together, changing along the effective resistances of the network, but is a widely used technique in electric circuit studies. These observations can have a significant impact in both experimental and analytical studies, as they can help us address common and recurring problems in peer-to-peer networks such as content replication, community detection and load balancing issues.

3.2 Tackling Problems in Peer-to-Peer Applications

Content Replication. To study the effects of content replication on the query lookup time one can use the hypernode technique: Nodes that possess the replicated content cannot be distinguished in terms of the desired outcome, which is simply to find the queried information, so they can be considered as a single hypernode and destination of the walker. Using the effective resistance it is easy to compare different replication schemes by the amount of closeness between the query issuer and nodes possessing the queried information.

Overloading Issues. Since the biased random walker tends to move on least resistance paths (instead of geodesic paths), it is reasonable to use weighted shortest paths in the calculation of the network diameter. In that case, a node affects the network diameter only if it is situated on a maximum weighted shortest path and through the weights of the edges that participate in this path.

Consider the case of node overloading in a peer-to-peer network, due to excess network traffic: An effective way for a node v to declare that it is overloaded is to change its edge weights to some high values and avoid servicing its neighbors, since the high communication costs will force them to try routing messages through other nodes. We don't know if this node participates on a maximum weighted shortest path and through which edges (if any), but we can estimate the impact of its decision to change edge weights by considering the rest of the network as a hypernode h and node v connected with it in parallel by its edges. We can then compute the difference achieved by changing the edge weights to their high values, and make some observations about the new network diameter.

As an example consider the random walker that navigates through high degree nodes by setting weights $w_{xy} = k_x \cdot k_y$, where k_x and k_y are the degrees of nodes x and y respectively. It is easy to see that the effective resistance between hypernode h and node v is $R_{hv} = \frac{1}{\sum_{i \in \Gamma_v} w_{vi}} = \frac{1}{k_v \sum_{i \in \Gamma_v} k_i}$. If we use unit weights as high values, we get that the difference in effective resistance between overloaded and normal states is

$$\Delta R_{hv} = \frac{1}{k_v} - \frac{1}{k_v \sum_{i \in \Gamma_v} k_i} = \frac{1}{k_v} \left(1 - \frac{1}{\sum_{i \in \Gamma_v} k_i} \right) \quad (3)$$

and that is the effect on the network diameter inflicted by the overloaded node v . We can see that this difference increases for lower degree nodes and larger neighborhood sizes (due to their greater criticality as bridges). Note that if nodes can recover from their overloaded state after some time, this technique can be useful in showing *self-regulating* properties of peer-to-peer networks on the diameter size.

Fault Tolerance. Node failures should increase the expected query lookup times, so fault tolerance can be studied by looking at the comparative growth of both effective resistance and mean first-passage time. During node removal special consideration should be given to the connectivity of the resulting network, a prerequisite in order to compute the framework metrics. Checking for connectivity in our framework can be easily done through the algebraic connectivity, its computation being a by-product of the matrix manipulations necessary to compute the framework metrics (table [II](#)).

Edge Selection/Dropping Policies. Our framework can be easily used to study how local decisions affect the network as a whole and correlate local and global phenomena in a quantifiable way [\[26\]](#). Such an example can be the global effect of local edge dropping policies, i.e. selection of neighbors in the overlay. In most networks peers drop some connections (either periodically or on demand) in favor of others, in order to maximize some parameter such as the average latency, bandwidth utilization or an interest-based metric. A possible outcome of a study like this could be the specification of a local edge dropping policy with the least detrimental effect on the network.

Computational Demands. Inverting a large, dense matrix such as the Laplacian may educe a high computational cost. Also, the fact that the Laplacian needs global information about the network in order to produce metrics such as the mean commute time between even two nodes, coupled with its size (equal to the network size), may be considered as restrictive regarding its usage in an experimental setting. However, as we have seen in this section, this effect is mitigated by the fact that we can compute several metrics at each inversion, as well as the existence of useful by-products such as the algebraic connectivity. Furthermore, parallel algorithms for graph related studies relevant to these (e.g. shortest path problems) are the subject of revived and ongoing research in the scientific community [\[27\]](#). The advances at this front, together with the fact that increasing degree of parallelism is continuously becoming available even in commonly used equipment, enable large sizes of networks to be analyzed. In the experimental study of this paper, which is given to exemplify the use of the framework, we used a SIMD multicore processor (GPU) with 32 cores in order to run the necessary elementary matrix operations in parallel.

4 Example Experimental Study

We illustrate the use of several of the methods outlined above in a concrete experimental study of example random walks and network types. For these studies we outline insights than can be gained and conclusions that can be drawn using our framework.

In particular, for these experiments we consider a random walker such as the one described in section 3, which uses node degrees as a navigational metric; i.e. the queries navigate strongly towards high degree nodes. Regarding the overlay topology, we use

synthetic data of networks with power-law degree distribution, based on the Barabási-Albert preferential attachment procedure [28], that produce a clear, small set of high degree nodes (past studies have linked peer-to-peer networks, e.g. the Gnutella network, with heavy tail degree distributions such as the power-law distribution [8][29]).

The strategies we used for the experiments have common elements and are given in table 2. The experiments were conducted over 50 graph instances of size 1000. Much of the computational load was lifted by executing elementary matrix operations in parallel on a nVidia GPU having 32 cores, using the CUDA library [30].

Table 2. Strategies used in experimental studies

Content Replication/Fault Tolerance Strategies

Strategy	Name	Description
Highest degree node	HIGHREP/HIGHFAIL	Replication to (failure of) nodes in decreasing highest degree sequence
Lowest degree node	LOWREP/LOWFAIL	Replication to (failure of) nodes in increasing lowest degree sequence
Random node	RANDREP/RANDFAIL	Replication to (failure of) random nodes

Edge Dropping Policies

Strategy	Name	Description
Highest resistance edge drop	HIGHRESD	Highest resistance edge drop from node under consideration
Lowest resistance edge drop	LOWRESD	Lowest resistance edge drop from node under consideration
Random edge drop	RANDD	Random edge drop from node under consideration
Highest voltage neighbor drop	HIGHVOLTD	Drop of edge between node and neighbor of highest voltage

4.1 Content Replication Example

For the experiments we use a sample query, initiated in a fixed node, that tries to locate a piece of information that initially can be found only in one node. At each step we replicate this information to another node, using one of the strategies described in table 2 with all nodes in possess of the information forming a hypernode. As we described in section 3, we measure the effective resistance and the mean first-passage time between the query issuer and this hypernode of nodes possessing the requested information.

Observations/Conclusions. As can be seen in figure 2, the effective resistance and mean first-passage times tend to decrease as we replicate content into more nodes. This is only natural since the desired information can be found more easily as replication progresses, and distances in the network decrease. Furthermore, we expected the HIGHREP strategy to have greater impact since it targets the high degree nodes used preferentially by the random walker and indeed this is the case, followed by RANDREP and LOWREP. However, we also note a seemingly counter-intuitive behavior of HIGHREP: while mean first-passage time continues to drop, the effective resistance starts rising

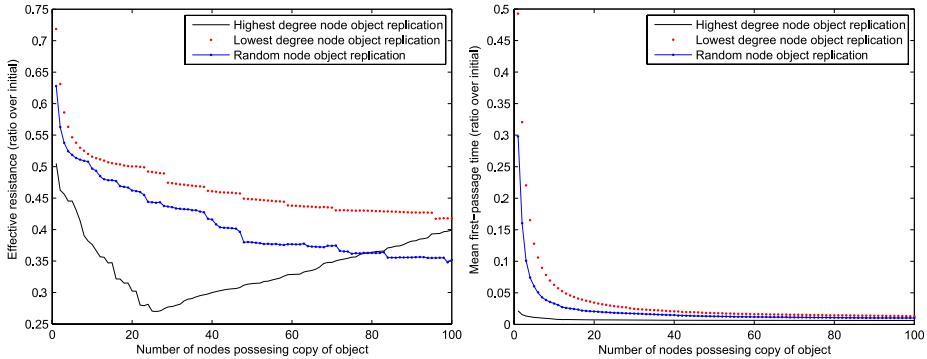


Fig. 2. Effective resistance and mean first-passage time between query source and destination, in the case of multiple object copies in the network

after some replications. This can be explained by recalling that the effective resistance is proportional to the mean commute time, so while the time needed to traverse the desired direction of the query (mean first-passage time of figure 2) is dropping, the time needed for the reverse direction can be increasing even more so. In a way, replication turns the growing hypernode (destination of the query) into a "gravity well".

4.2 Fault Tolerance Example

In this scenario we remove from each network 90 nodes in batches of 30, according to the strategies described at table 2 and we aim to study the detrimental effect of node failures. As we described in section 3, we expect distances in the network to grow so we compute the comparative growth of both the effective resistance and mean first-passage time at the end of each batch, all the while checking for connectivity through the algebraic connectivity. The means over all pairs, for all strategies and experiments, can be found in figure 3.

Observations/Conclusions. Observe that the HIGHFAIL strategy is the most destructive of all, raising the mean value of the effective resistance by a factor of 19. This was expected since it destroys high degree nodes and these are preferred by the walker. The LOWFAIL strategy has negligible effect on the network, since the low degree nodes are rarely used for routing purposes. The RANDFAIL strategy is proven equally powerless but slightly more destructive, since it may target a high degree node but low degree nodes are the majority.

However, the HIGHFAIL strategy behaves differently in various regions of the network. To investigate this phenomenon we studied more closely a subset of the previous results concerning two communication groups of interest, one of very high degree to very low degree nodes and its symmetric (figure 4). Although the qualitative effect of each strategy is the same in each communication group, the above mentioned unbalance is clearly shown. Again, this was expected since the walker navigates towards nodes of high degree and it appears that the highest degree nodes were connected to low degree

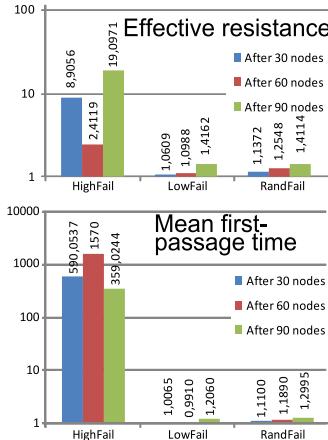


Fig. 3. Comparative growth of mean first-passage times and effective resistance for different failure strategies

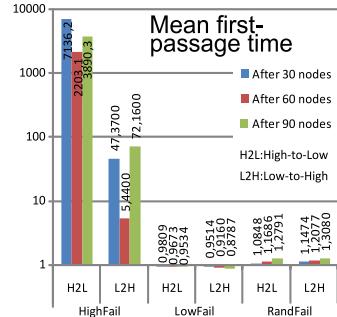


Fig. 4. Effect of failures on groups of interest

ones. Having lost these bridges it is becoming increasingly difficult for the walker to make the transition from high to low degree nodes. Another point of interest is the behavior of the HIGHFAIL strategy during the second batch of deletions, where the metric values drop. It appears that with the exception of a small number of initial deletions, subsequent deletions improve the mean first-passage and commute times, and only quite later these metrics begin to decline again. This is a direct consequence of the walker's memorylessness: he cannot track his previous steps on the query path and therefore falls into cycles, which can be quite large when involving high degree nodes since the latter constitute local "gravity centers" for the walker. So, by removing them with the HIGHFAIL strategy we remove these cycles and we actually help the walker to reach its destination faster. Of course this phenomenon is paired with the previously mentioned difficulty of high degree nodes reaching low degree ones, and the end result is constrained growth of mean first-passage and commute times. It appears that navigating through high degree nodes can be a double edged tool when dealing with a memoryless random walker.

4.3 Edge Dropping Policies Example

In a similar way to the study of fault tolerance, we expect the distances in the network to grow as we drop more edges, so we study the comparative growth of both the effective resistance and mean first-passage time. As we will see some strategies succeed in dropping slightly these metrics, compared to their original values. In order to study the global effect of the local edge dropping policies we employ the following experimental procedure: For each graph we are instantiating we select a set of 100 nodes that will perform the specified edge drop policy. Then we apply the policy to the set of nodes in batches of 25 and after each batch we calculate the mean values of the effective resistance and mean first-passage time between all node pairs in the network, divided with

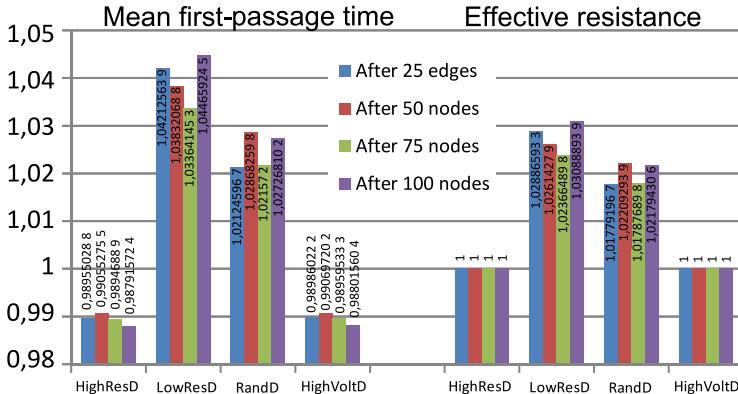


Fig. 5. Comparative changes of mean first-passage times and effective resistance for different edge dropping strategies

their original values to show the relative growth or shrinkage. We added one more strategy (HIGHVOLTD) to the ones used previously, in order to see whether better results can be achieved by using global information to decide on the edge we should drop. Note that here it is safe to look only at mean values over all pairs, since values of the metrics under study vary significantly less compared to the values of previous experiments.

Observations/Conclusions. Observe that the HIGHRESD strategy targets edges with high resistance, therefore connecting to lower degree nodes, in contrast to the LOWRESD strategy which targets edges with higher degree nodes. Consistent with the results regarding fault tolerance, we also observe that the LOWRESD strategy has the most detrimental effect, with HIGHRESD even having a slightly positive result and RANDD performing between these two. It may be worth to take a special look at the HIGHRESD and HIGHVOLTD strategies, which perform quite similarly and even succeed in maintaining the effective resistance to the same mean value as it was originally. However, HIGHVOLTD has a high computational cost (needing global information). Using this framework we are able to safely conclude that implementing a costly global policy gives us a negligible gain in performance, compared to the simple and local policy of HIGHRESD which also gives the best results over all strategies.

5 Conclusions

In this paper we introduced REPO, a framework for the analysis and systematic study of random walks on peer-to-peer networks. To illustrate the possible uses of REPO, we presented a basic set of analytical tools and a brief overview of recurring problems in peer-to-peer networks that can be studied experimentally. Our aim was not to enumerate exhaustively all potential application areas but rather to illuminate the multitude of connections between classic electric theory, random walks and peer-to-peer networks and present some examples of the framework's intuitiveness.

We present analytical tools that enable us to reason about the robustness achieved by different random walks on the same network and study the effects of overloading

nodes to the peer-to-peer overlay. We also relate local policies with global phenomena observed on the network, and reason on the effectiveness of the policies under study. Finally, we draw conclusions on the interaction between random walks, their underlying peer-to-peer topologies and how they affect the query efficiency, especially using content replication techniques or under the presence of faulty nodes. We observe that the performance of the walker depends heavily on properties of the overlay. For example, the promising performance results reported by recent work [5][6] do not seem to transfer directly to other overlays, such as the power-law networks of our example study.

Other issues of peer-to-peer research that can benefit from our framework paradigm are load balancing techniques, as well as construction techniques that utilize specialized random walks. Algebraic connectivity and effective resistance can express structural properties quantitatively and be employed i.e. during the construction phase to steer it towards desirable values.

Acknowledgements

We are thankful to Daniel Cederman for his help with using the CUDA library in our experiments.

References

1. Balakrishnan, H., Kaashoek, F.M., Karger, D., Morris, R., Stoica, I.: Looking up data in p2p systems. *Commun. ACM* 46(2), 43–48 (2003)
2. Adamic, L.A., Lukose, R.M., Puniyani, A.R., Huberman, B.A.: Search in power-law networks. *Phys. Rev. E* 64, 46135 (2001)
3. Ata, S., Murata, M., Gotoh, Y.: Replication methods for enhancing search performance in peer-to-peer services on power-law logical networks. In: *Performance and Control of Next-Generation Communications Networks*. SPIE, vol. 5244, pp. 76–85 (2003)
4. Gkantsidis, C., Mihail, M., Saberi, A.: Random walks in peer-to-peer networks. In: *Proc. of 23rd Annual Joint Conf. of the IEEE Computer and Communications Societies (INFOCOM 2004)*, March 2004, vol. 1 (2004)
5. Sarshar, N., Boykin, P.O., Roychowdhury, V.P.: Percolation search in power law networks: making unstructured peer-to-peer networks scalable. In: *Proc. of the 4th International Conf. on Peer-to-Peer Computing*, pp. 2–9 (2004)
6. Zhaoqing, J., Jinyuan, Y., Ruonan, R., Minglu, L.: Random walk search in unstructured p2p. *J. Syst. Eng.* 17(3), 648–653 (2006)
7. Chockler, G., Melamed, R., Tock, Y., Vitensberg, R.: Spidercast: a scalable interest-aware overlay for topic-based pub/sub communication. In: *Proc. of 2007 inaugural international conf. on Distributed event-based systems (DEBS 2007)*, pp. 14–25. ACM, New York (2007)
8. Fraigniaud, P., Gauron, P., Latapy, M.: Combining the use of clustering and scale-free nature of user exchanges into a simple and efficient p2p system. In: Cunha, J.C., Medeiros, P.D. (eds.) *Euro-Par 2005. LNCS*, vol. 3648, pp. 1163–1172. Springer, Heidelberg (2005)
9. Voulgaris, S., Kermarrec, A.M., Massoulie, L.: Exploiting semantic proximity in peer-to-peer content searching. In: *Proc. of 10th IEEE International Workshop on Future Trends of Distributed Computing Systems (FTDCS 2004)*, pp. 238–243 (2004)
10. Hughes, B.: *Random Walks and Random Environments: Random Walks*, vol. 1. Clarendon Press, Oxford (1995)

11. Klein, D.J., Randić, M.: Resistance distance. *J. Math. Chem.* 12(1), 81–95 (1993)
12. Bui, A., Sohier, D.: How to compute times of random walks based distributed algorithms. *Fundamenta Informaticae* 80(4), 363–378 (2007)
13. Sohier, D., Bui, A.: Hitting times computation for theoretically studying peer-to-peer distributed systems. In: Proc. of the 18th International Parallel and Distributed Processing Symposium (2004)
14. Telcs, A.: *The Art of Random Walks*. Springer, Heidelberg (2006)
15. Lv, Q., Cao, P., Cohen, E., Li, K., Shenker, S.: Search and replication in unstructured peer-to-peer networks. In: Proc. of the 16th International Conf. on Supercomputing (ICS 2002), pp. 84–95. ACM Press, New York (2002)
16. Barnett, S.: *Matrices: Methods and Applications*. Oxford University Press, Oxford (1990)
17. Fiedler, M.: Algebraic connectivity of graphs. *Czechoslovak Math. J.* 23, 298–305 (1973)
18. de Abreu, N.M.: Old and new results on algebraic connectivity of graphs. *Linear Algebra and its Applications* 423(1), 53–73 (2007)
19. Chung, F.: *Spectral Graph Theory*. CBMS Regional Conf. Series in Mathematics, vol. 92. AMS (1997)
20. Doyle, P.G., Snell, L.J.: *Random Walks and Electrical Networks*. Mathematical Association of America (December 1984)
21. Chandra, A.K., Raghavan, P., Ruzzo, W.L., Smolensky, R.: The electrical resistance of a graph captures its commute and cover times. In: Proc. of the 21st Annual ACM Symposium on Theory of Computing (STOC 1989), pp. 574–586. ACM Press, New York (1989)
22. Alexander, C., Sadiku, M.: *Fundamentals of Electric Circuits*. McGraw-Hill, New York (2006)
23. Fouss, F., Pirotte, A., Renders, J.M., Saerens, M.: Random-walk computation of similarities between nodes of a graph with application to collaborative recommendation. *IEEE Trans. Knowl. Data Eng.* 19(3), 355–369 (2007)
24. Brin, S., Page, L.: The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.* 30(1-7), 107–117 (1998)
25. Olfati-Saber, R., Murray, R.: Consensus problems in networks of agents with switching topology and time-delays. *IEEE Trans. Autom. Control* 49(9), 1520–1533 (2004)
26. Georgiadis, G., Kirovius, L.: Lightweight centrality measures in networks under attack. In: Proc. of European Conf. on Complex Systems, ECCS (2005)
27. Madduri, K., Bader, D.A., Berry, J.W., Crobak, J.R., Hendrickson, B.A.: Multithreaded algorithms for processing massive graphs. In: Bader, D.A. (ed.) *Petascale Computing: Algorithms and Applications*. Chapman & Hall/CRC Computational Science Series, Boca Raton (2008)
28. Albert, R., Barabási, A.L.: Statistical mechanics of complex networks. *Rev. Mod. Phys.* 74(1), 47–98 (2002)
29. Ripeanu, M., Foster, I., Iamnitchi, A.: Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. *IEEE Internet Comput.* 6(1), 50–57 (2002)
30. NVIDIA Corporation nvidia.com/cuda: NVIDIA CUDA Programming Guide. 2.0b edn. (2008)

SiMPSON: Efficient Similarity Search in Metric Spaces over P2P Structured Overlay Networks

Quang Hieu Vu^{1,2}, Mihai Lupu³, and Sai Wu⁴

¹ Imperial College London, UK
qhv@doc.ic.ac.uk

² Institute for Infocomm Research, Singapore
qhv@i2r.a-star.edu.sg

³ Information Retrieval Facility, Austria
m.lupu@ir-facility.org

⁴ National University of Singapore, Singapore
wusai@comp.nus.edu.sg

Abstract. Similarity search in metric spaces over centralized systems has been significantly studied in the database research community. However, not so much work has been done in the context of P2P networks. This paper introduces SiMPSON: a P2P system supporting similarity search in metric spaces. The aim is to answer queries faster and using less resources than existing systems. For this, each peer first clusters its own data using any off-the-shelf clustering algorithms. Then, the resulting clusters are mapped to one-dimensional values. Finally, these one-dimensional values are indexed into a structured P2P overlay. Our method slightly increases the indexing overhead, but allows us to greatly reduce the number of peers and messages involved in query processing: we trade a small amount of overhead in the data publishing process for a substantial reduction of costs in the querying phase. Based on this architecture, we propose algorithms for processing range and kNN queries. Extensive experimental results validate the claims of efficiency and effectiveness of SiMPSON.

1 Introduction

Similarity search is the process of retrieving data objects that are analogous to the query object based on a given criterion. This kind of search is particularly important to applications such as multimedia information retrieval, computer vision, or biomedical databases. Extensive work has been done to support similarity search in centralized systems [6][12]. A straight-forward solution is to employ conventional multi-dimensional index structures/methods such as [3][4][11][22] to index objects. Since these index structures/methods only work in vector spaces, other index structures/ methods, specific for general metric spaces, have been invented [8][13].

Even though there have already been some proposals to support similarity search in P2P systems [1][10][15][17][18], they only provide solutions that adapt conventional centralized index structures/methods to the P2P environment. In their design, they do not take into account the fact that there are often a large number of objects in a P2P system, and hence indexing every object will incur a high cost. To alleviate this problem, a viable solution is to cluster objects and index only the resulting clusters. However, it

is not straight-forward to index clusters in P2P systems, and existing solutions can be improved upon. Most notably with respect to the number of peers involved in each query request. Also, using a structured overlay rather than an unstructured one should increase the efficiency of the indexing and querying processes.

As a result, in this paper, we propose an alternative solution, which involves the use of an extended version of iDistance [13] for indexing object clusters. We introduce SiMPSON¹: a P2P system supporting similarity search where each peer node first clusters its own data using a clustering algorithm such as K-Means. The resulting data clusters are then mapped to one-dimensional values by our extended version of iDistance. Finally, these one-dimensional values are indexed into a structured P2P network. Based on this architecture, we present algorithms for processing range and kNN queries. We also show how to estimate the initial radius of a kNN query in our system. To summarize, this paper makes the following major contributions:

- We introduce a new index architecture to support similarity search in metric spaces over P2P systems.
 - By having two indices for each cluster instead of one, our indexing method is able to significantly reduce the number of peers involved in each query.
- We present algorithms to support efficient range and kNN search. For kNN search, we also present and compare three different methods to estimate the initial radius of the range query that will retrieve the desired k data items.
- We have conducted experiments over PlanetLab [7], a testbed for large-scale distributed systems, to validate the effectiveness and efficiency of our proposal and compare against SimPeer, the current state of the art

The rest of the paper is organized as follows. In Section 2 we present related work. In Section 3, we introduce SiMPSON’s indexing architecture. In Section 4 we discuss how to use indices to process range queries and kNN queries. Finally, we present the experimental study in Section 5 and conclude in Section 6.

2 Related Work

Similarity search in centralized systems has been extensively studied in the database research community. Several index structures have been proposed, focusing on pruning of the search space at the query time. These structures are generally categorized into two classes: tree-based indexing in which the data space is hierarchically divided into smaller subspaces to form a tree structure [3,4,8,11,12] and mapping-based indexing in which objects are mapped to one-dimensional values and a one-dimensional indexing structure is used to index these mapping values [19,13].

In P2P systems, similarity search is often supported by adapting centralized index structures. DPTree [17] and VBI-Tree [15] are typical examples of P2P systems adapting tree-based indexing. These systems are designed as frameworks that can adapt different kinds of multi-dimensional tree index structures such as R-Tree [11] and M-Tree [8]. On the other hand, MCAN [10] and M-Chord [18] are models of P2P systems adapting mapping-based indexing. These systems employ a pivot-based technique that

¹ Similarity search in Metric spaces over P2P Structured Overlay Networks.

maps objects to values in a multi-dimensional space (M-CAN) or a one-dimensional space (M-Chord). These values are subsequently indexed to CAN [21] or Chord [16] respectively. In other approaches, SWAM [1] proposes a family of access methods for similarity search by building an overlay network where nodes with similar content are grouped together while LSH forest [2] utilizes locality-sensitive hash functions to hash similar objects to the same bucket and indexes the bucket on a prefix-tree structure.

A common problem to all of the above systems is that they incur a high resource cost by attempting to index all the data items. Addressing this issue, SimPeer [9], a super-peer based system, the closest to our work, suggests that peers should summarize their objects before indexing and that only these summaries be indexed to the system. In this way, these methods can significantly reduce the cost of indexing. SimPeer uses a set of globally known reference points which divide the entire data space into partitions. To index a particular cluster C_x , SimPeer first assigns this cluster to a partition P_i - the one whose reference point is closest to the center of the cluster. It then maps the *farthest* point of C_x to a one-dimensional index value based on the reference point O_i of partition P_i . To process a range query, for each partitions P_i which is formed by a reference point O_i and intersects with the query, SimPeer has to search from the nearest point of the query to O_i to the boundary of P_i . This is a weakness of SimPeer because the search space depends on the nearest point of the query to O_i . Even though the query size is small, if the nearest point of the query is close to O_i , the system still needs to search a large space. Additionally, since partitions need to be enlarged to encompass the farthest point (the index) of each cluster, large overlaps among data partitions may be generated, resulting in unnecessary replication of indices. SiMPSON is different from SimPeer in two main aspects: (1) it is designed to alleviate the problems of SimPeer we just described and (2) it is designed for structured P2P systems, in order to optimally use the network's resources.

3 Indexing Architecture

In SiMPSON, each peer node first clusters its own data using a clustering algorithm. After that, the resulting clusters are indexed directly to the overlay network, which can be any structured P2P system supporting single-dimensional range queries. This is done using our extension of iDistance [13] to allow mapping a data cluster to one-dimensional values for indexing. Let $S_P = \{P_1, P_2, \dots, P_n\}$ be a set of data partitions used by the iDistance method and $S_O = \{O_1, O_2, \dots, O_n\}$ be the corresponding reference points of these partitions. Let C_x be a data cluster. In this section, we introduce the way to index C_x in the overlay network according to S_P and S_O . For simplicity, we require that C_x does not cover any reference point in S_O . If C_x covers a reference

Table 1. Table of symbols

Symbol	Definition
key_O	the mapping value of a reference point O in one-dimensional space
$P(O, r)$	a data partition whose reference point and radius are O and r
$C(K, R)$	a data cluster whose center point and radius are K and R
$I_S(C, O)$	a starting index of a cluster C with respect to a reference point O
$I_E(C, O)$	an ending index of a cluster C with respect to a reference point O
$\zeta(N)$	a covering cluster counter of a peer node N

point, it can be easily split into smaller clusters. For easy reference, Table I shows the frequent symbols used throughout this paper.

3.1 Definitions

Each cluster $C_x(K_x, R_x)$ is indexed with respect to a reference point O_i . Each index entry keeps information on the cluster center, K_x , the cluster's radius, R_x , the number of data items inside the cluster, D_x and the IP address of the peer publishing the cluster, IP_x . We denote an index by $I(C_x, O_i) = \{key_x, K_x, R_x, D_x, IP_x\}$. There are two types of cluster indices, which are defined as follows.

Definition 1. Starting Index: $I_S(C_x, O_i)$ and Ending Index: $I_E(C_x, O_i)$

- A Starting Index of a cluster $C_x(K_x, R_x)$ with respect to a reference point O_i , denoted as $I_S(C_x, O_i)$, is a cluster index $I(C_x, O_i)$ where $key_x = key_{O_i} + [dist(K_x, O_i) - R_x]$.
- An Ending Index of a cluster $C_x(K_x, R_x)$ with respect to a reference point O_i , denoted as $I_E(C_x, O_i)$, is a cluster index $I(C_x, O_i)$ where $key_x = key_{O_i} + [dist(K_x, O_i) + R_x]$.

Intuitively, the Starting and Ending Index values of a data cluster C_x are respectively based on the smallest and largest distance between any point of the sphere representing the cluster and the reference point O_i . Besides these two types of cluster indices, we also have a special parameter called Covering Cluster Counter, which is defined as follows.

Definition 2. Covering Cluster Counter: $\zeta(N_i)$

A covering cluster counter of a peer node N_i , denoted as $\zeta(N_i)$, is a value which counts the number of clusters whose Starting Index value is smaller than the minimum value of which N_i is in charge, and Ending Index value is greater than the maximum value of which N_i is in charge.

Figure II shows an example of starting indices and ending indices of two clusters $C_x(K_x, R_x)$ and $C_y(K_y, R_y)$, as well as a demonstration of how covering cluster counters of nodes are calculated. In this figure, the two indexing clusters fall into the data partition $P_i(O_i, r_i)$ whose mapping range of values is stored by nodes $N_1, N_2, N_3, N_4, N_5, N_6$,

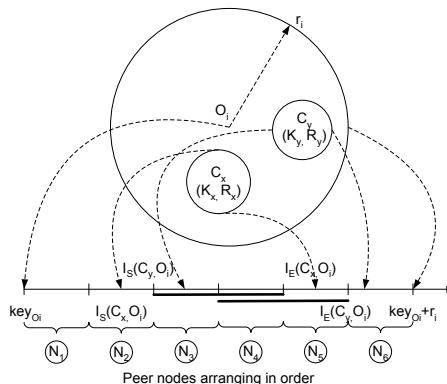


Fig. 1. An example of starting index, ending index, and covering cluster counter

N_4 , N_5 , and N_6 . The starting index of $C_x(K_x, R_x)$ is mapped to the value $key_{O_i} + [dist(K_x, O_i) - R_x]$ belonging to node N_2 . The ending index of $C_x(K_x, R_x)$ is mapped to the value $key_{O_i} + [dist(K_x, O_i) + R_x]$ belonging to node N_5 . The starting index of $C_y(K_y, R_y)$ is mapped to the value $key_{O_i} + [dist(K_y, O_i) - R_y]$ belonging to node N_3 . The ending index of $C_y(K_y, R_y)$ is mapped to the value $key_{O_i} + [dist(K_y, O_i) + R_y]$ belonging to node N_6 . Since both $C_x(K_x, R_x)$ and $C_y(K_y, R_y)$ cover the range of values of N_4 , $\zeta(N_4) = 2$. On the other hand, since the ranges of values stored by N_3 and N_5 are covered by either $C_x(K_x, R_x)$ or $C_y(K_y, R_y)$, $\zeta(N_3) = 1$ and $\zeta(N_5) = 1$.

3.2 Indexing Method

For each cluster $C_x(K_x, R_x)$, SiMPSON considers two cases to index the cluster.

1. $C_x(K_x, R_x)$ fits in a data partition $P_i(O_i, r_i)$ as in Figure 2(a) (if C_x fits in more than one data partition, the *nearest* data partition is selected to index C_x). In this case, the system creates a starting index $I_S(C_x, O_i)$ and an ending index $I_E(C_x, O_i)$ for the cluster. Additionally, assume that $I_S(C_x, O_i)$ is stored at node N_p and $I_E(C_x, O_i)$ is stored at node N_q , for all nodes falling between N_p and N_q in the linear order, the system increases their counter ζ by one. As in Figure 2(a), indexing the cluster C_x causes insertions of $I_S(C_x, O_i)$ at node N_3 and $I_E(C_x, O_i)$ at node N_5 , and an increasing of $\zeta(N_4)$.
2. $C_x(K_x, R_x)$ does not fit in any data partition as in Figure 2(b). In this case, the system creates a starting index $I_S(C_x, O_i)$ for each partition $P_i(O_i, r_i)$, which intersects with the cluster. Additionally, assume that $I_S(C_x, O_i)$ is stored at node N_i . For all nodes located after N_i in the linear order, and whose ranges of values fall into the index interval of the data partition P_i , the system increases their counter ζ by one. Note that no ending indices are created in this case. As in Figure 2(b), since the cluster C_x intersects with two partitions $P_i(O_i, r_i)$ and $P_j(O_j, r_j)$, indexing C_x causes insertions of $I_S(C_x, O_i)$ at node N_3 and $I_S(C_x, O_j)$ at node N_8 , and an increase of $\zeta(N_4)$ and $\zeta(N_5)$.

In this way of cluster indexing, there are no cases where the system needs to enlarge data partitions to fit large clusters, as it was the case in SimPeer. As a result, compared

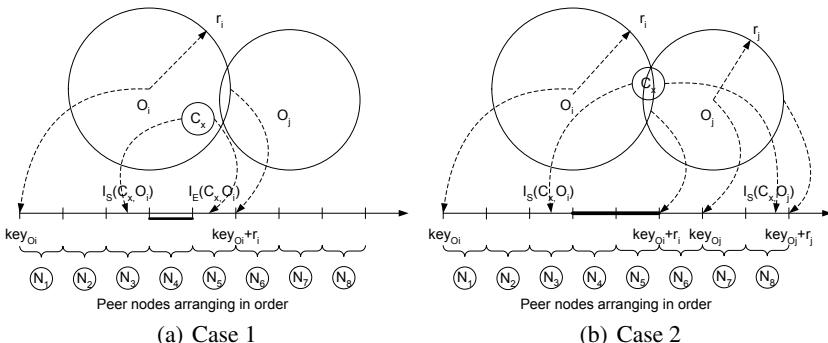


Fig. 2. Cluster indexing

to SimPeer, our method significantly reduces the amount of overlaps between partitions and, as a direct consequence, the number of redundant messages between peers.

3.3 Index Management

When a data cluster C_x is published to the system, if the cluster falls into the first case, the system finds the node N_p to index $I_S(C_x, O_i)$. After that, following right adjacent links of N_p and following nodes, the system increases the covering cluster counters of nodes it goes through until it reaches the node N_q to index $I_E(C_x, O_i)$. The cost of this process is $O(\log N + \delta)$ steps, where δ is the number of nodes falling between the N_p and N_q . Similarly, if the cluster falls into the second case, the system needs to find the node N_i to index $I_S(C_x, O_i)$ for every partition $P_i(O_i, r_i)$, which intersects with the clusters. After that, right adjacent links are followed to increase the covering cluster counters of neighbor nodes until the node holding the maximum index value of P_i is found. Even though, there may be multiple data partitions intersecting with the cluster, and hence multiple starting indices may need to be inserted, the nodes N_i, N_j, \dots, N_k in charge of them can be found in parallel. As a result, the cost of this process is also bounded at $O(\log N + \delta')$ steps, where δ' is the maximum number of nodes falling between a node N_i, N_j, \dots, N_k and the node holding the maximum index value of the corresponding data partitions P_i, P_j, \dots, P_k . When a data cluster is removed from the system, an opposite process is executed to remove indices of the cluster and reduce the covering cluster counters of related nodes.

4 Query Processing

Before presenting algorithms for processing range and kNN queries, we introduce an important observation, which is used to filter out unwanted clusters in query processing.

Observation 1. *A data cluster $C_x(K_x, R_x)$ intersects with a range query $Q_R(q, r)$ if and only if $\text{dist}(K_x, q) \leq R_x + r$ ($\text{dist}(K_x, q)$ is the distance between K_x and q).*

4.1 Range Query

SiMPSON processes a range query $Q_R(q, r)$ in three steps.

1. First, the node issuing the query determines all data partitions which intersect with the query. According to Observation 1, a partition $P_i(O_i, r_i)$ intersects with $Q_R(q, r)$ if $\text{dist}(O_i, q) \leq r_i + r$. After that, for each intersecting partition $P_i(O_i, r_i)$, let $\text{key}_{\min} = \text{key}_{O_i} + \text{dist}(O_i, q) - r$ and $\text{key}_{\max} = \text{key}_{O_i} + \text{dist}(O_i, q) + r$ be respectively the mapping of the nearest point and the farthest point in Q_R to a one dimensional value (if the farthest point falls outside P_i , $\text{key}_{\max} = \text{key}_{O_i} + r_i$). The node then creates a one-dimensional range query $Q'_R(\text{key}_{\min}, \text{key}_{\max})$ to retrieve all indices, which fall in this search region, since clusters of these indices may intersect with the search region.

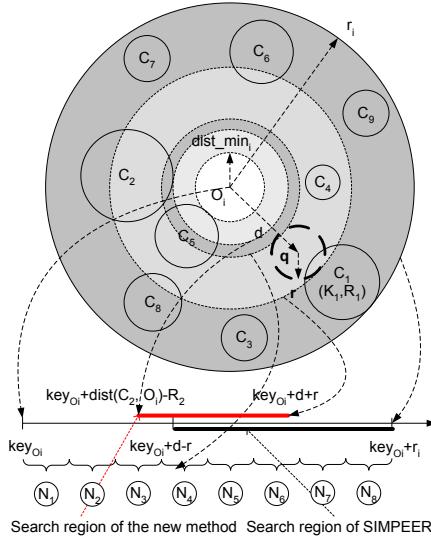


Fig. 3. Query Processing

2. The above step does not consider the case where a data cluster $C_x(K_x, R_x)$ covers the search region. In this case, since both the starting index and the ending index of $C_x(K_x, R_x)$ fall outside the one-dimensional search region, further search is needed. Let S_N be the set of nodes storing indices in the range $[key_{min}, key_{max}]$; S_I be the set of indices stored at nodes in S_N ; and N_i be the node in charge of key_{min} . Set $T = \zeta(N_j) - |\{I_E(C_x, O_i), I_E(C_x, O_i) \in S_I \text{ and } I_S(C_x, O_i) \notin S_I\}|$. Since T is actually the number of clusters whose indices cover the one-dimensional range of values managed by nodes in S_N , if $T = 0$, no further search is needed. If $T > 0$, a special query request $Q''_R(q, r, T)$ is sent to the left adjacent node N_j of N_i . N_j then searches all starting indices it is storing. For each starting index $I_S(C_x, O_i)$, if $key_x + 2 \cdot R_x > key_{max}$, N_j returns the index to the query requester and decreases T by one. At the end of this step, if $T = 0$, the search process stops. Otherwise, N_j continues to forward Q''_R to its left adjacent node and so on.
3. Finally, based on result indices received from the above steps, the query requester node finds data clusters which actually intersect with the query by checking the distance between q and the center of the data clusters as in Observation 1. It then sends Q only to the nodes owning the intersecting clusters.

An example of range query processing is shown in Figure 3. For simplicity, we assume that there is only one data partition $P_i(O_i, r_i)$, which intersects with the query $Q_R(q, r)$. As a result, in the first step, the node issuing the query searches for only one one-dimensional range query $Q'_R(key_{O_i} + d - r, key_{O_i} + d + r)$. The results of executing Q'_R should be $I_S(C_1, O_i), I_S(C_4, O_i), I_E(C_4, O_i), I_E(C_5, O_i), I_E(C_6, O_i)$, and $I_S(C_8, O_i)$, which are stored in three nodes N_4, N_5 , and N_6 . After that, in the second step, since T should be equal to 1, a special range query $Q''_R(q, r, 1)$ is sent to N_3 , the left adjacent node of N_4 . At N_3 , since $I_S(C_2, O_i)$ is satisfied the covering

condition, $T = T - 1 = 0$, and hence the process stops. Finally, by checking the intersecting condition, only the node publishing C_1 is selected to send the query $Q_R(q, r)$. Note that the overall one-dimensional search region of SiMPSON in this example is from $key_{O_i} + dist(C_2, O_i) - R_2$ to $key_{O_i} + d + r$ (thick red line, above the axis). This is shorter than in the case of SimPeer, where the one-dimensional search region is from $key_{O_i} + d - r$ to $key_{O_i} + r_i$ (thick black line, below the axis). In particular, if C_2 does not exist in this example, SiMPSON only needs to search exactly the mapping one-dimensional search region of the query, i.e. the range from $key_{O_i} + d - r$ to $key_{O_i} + d + r$. On the other hand, even in this case, the search region of SimPeer would still be the same. In general, in SiMPSON, an area outside the search region only needs to be searched if there exist clusters indexed in this area, which cannot be avoided.

4.2 kNN Query

In SiMPSON, there are two phases to process a kNN query $Q_{kNN}(q, k)$. The first phase is to estimate the initial radius ϕ of the query. After that, a range query $Q'_R(q, \phi)$ is executed. If there are at least k data items returned by executing Q'_R , the top k nearest neighbor data items of q are retrieved from returned results. Otherwise, if less than k data items are returned, the system increases the value of ϕ and re-executes the query. This process repeats until enough k data items are found. In the following, we first present a basic estimation method for the radius ϕ . We then introduce two improvements to this basic method, with the purpose of reducing the total number of messages exchanged between peers.

Basic Radius Estimation Method. To determine the initial radius, the node issuing the query first finds a data partition $P_i(O_i, r_i)$, which is the nearest to q (based on the distance between q and the origin O_i). After that, it sends a request to estimate the initial radius of the query, $E_{kNN}(q, k)$, to the node N_i , which is in charge of the one-dimensional value mapping of q based on O_i . If q falls outside $P_i(O_i, r_i)$ then N_i is the node in charge of the mapping value of the farthest point in P_i . Node N_i then retrieves its storing indices step-by-step, by increasing the distance between the index values and the mapping value of q . For a *retrieved index* $I(C_x, O_i)$, if its corresponding cluster covers q ($dist(K_x, q) \leq 2 \cdot R_x$), that index is marked as an *explored index*. We observe that for each index $I(C_x, O_x)$, a range query $Q_R(q, dist(q, K_x) + R_x)$ covers the entire cluster C_x . Consequently, executing this query should return at least D_x data items, which is the number of data items belonging to the cluster C_x . As a result, let $\sigma(D)$ be the total number of data items stored at clusters whose indices have been explored so far. N_i stops exploring indices when $\sigma(D) \geq k$. In this case, we take our estimate radius to be $\phi = dist_{max} = \max\{dist(q, K_x) + R_x\}$, for all explored indices $I(C_x, O_i)$. A range query $Q_R(q, \phi)$ should return at least $\sigma(D) \geq k$ data items.

Improved Radius Estimation Methods. In the previous section we calculated an upper bound on the radius of the range query that would retrieve k nearest data items. In this section, we try to reduce the radius, in order to reduce the number of peers we have to contact to retrieve the desired k items. We propose two methods: one based on what we

would call *local* data pertaining to node N_i and one based on information about clusters indexed at other nodes, but which cover the data range that is maintained by N_i .

Locally optimized method: One way to get a potentially better estimate of the radius of the range query that would retrieve exactly k nearest data items, is to take into account the proportion between the number of actually retrieved data items ($\sigma(D)$) and k . The idea is simple: if a range query with a radius $dist_{max}$ (area proportional to $dist_{max}^d$) retrieves $\sigma(D)$ items, then the radius of the range query that would retrieve exactly k items must be proportional to $\sqrt[d]{k/\sigma(D)}$, where d is the dimensionality. In particular, $\phi = dist_{max} \cdot \sqrt[d]{\frac{k}{\sigma(D)}}$. Note that while this method assumes that the data items are distributed uniformly inside the clusters and this is obviously not true, our experimental results in Section 5 show that it is a sufficient approximation to reduce the number of messages by at least 20%.

Globally optimized method: There is one piece of information that has not been taken into account in the previous two estimates: the clusters whose indices fall outside the range managed by node N_i . For them, our indexing method uses the cluster covering counter ζ . Now, let f be the ratio between the number of explored indices and the number of retrieved indices at N_i . We conjecture that the number of data items retrieved by a range query must also be a function of the clusters covering the current node, but, since we do not use all the local clusters either, we proportion the factor ζ by the fraction f . Assuming that data items are distributed uniformly among clusters, and knowing that a fraction f of the retrieved clusters gave us the $\sigma(D)$ documents, we extrapolate that ζ clusters that cover the current node will give us similar results. We consequently add $\zeta \cdot f \cdot \sigma(D)$ to the denominator of the fraction in the formula of the locally optimized method to obtain the new formula to calculate ϕ : $\phi = dist_{max} \cdot \sqrt[d]{\frac{k}{\sigma(D)(\zeta \cdot f + 1)}}$.

5 Experimental Study

To evaluate the performance of SiMPSON, we implemented a simulation system in Java and ran it on PlanetLab [7], a testbed for large-scale distributed systems. Because of our extensive experience with BATON [14], SiMPSON has been deployed on top of it. However, any structured overlay supporting range queries can be used. We tested the simulation system with three benchmark datasets: VEC [18] which contains 1 million 45-dimensional points synthesized by extension from a set of 11,000 digital image features; Covtype [5] which contains 581,000 54-dimensional points pertaining to geographic attributes like elevation, aspect, slope, distances, etc...; and Color Histograms [20] which contains 68,040 32-dimensional image features extracted from a Corel image collection. For comparison purposes, we also implemented the existing indexing method employed by SimPeer [9] - the current state of the art in the field.

Table 2. Experimental parameters

Parameters	Network size	Dimensionality	Query radius	k nearest point
Range of Values	100 - 4,000	2 - 32	0.01 - 0.05	5 - 25
Default Value	1,000	3	0.03	5

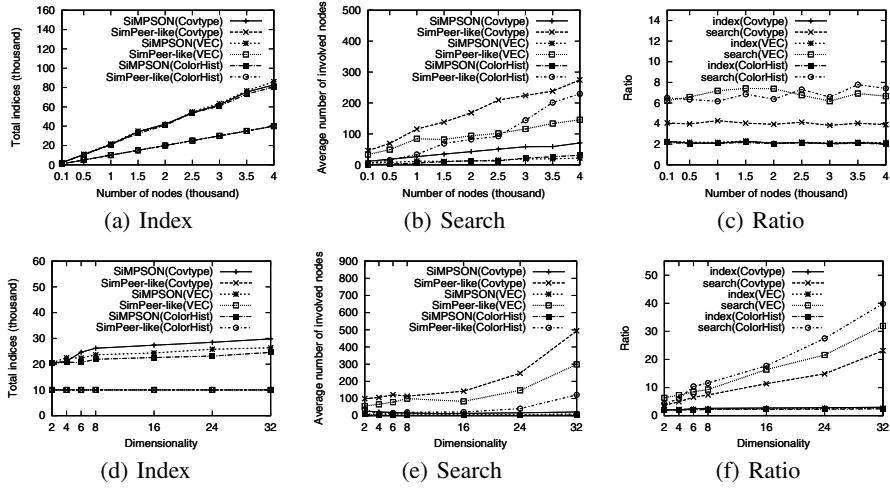


Fig. 4. Effects of network size (a, b, c) and Effects of dimensionality (d, e, f)

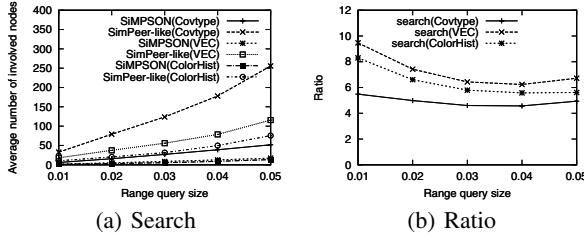
However, since this method is designed for super-peer based networks, it cannot be applied directly on a structured network. As a result, we created an additional server to maintain necessary statistics for this method (i.e. the minimum distance between indexing clusters and reference points and the boundaries of partitions since these values can be changed during the indexing process). We name this variant indexing method SimPeer-like. We compare SiMPSON and SimPeer-like on two important criteria: the number of indices created during index process and the number of nodes participating in query processing. Table 2 shows parameters, their default values and their ranges of values used in our experiments.

5.1 Effects of Network Size

We begin by evaluating the effects of increasing the network size. Figures 4(a) and 4(b) respectively show the total number of indices created by the system and the average number of nodes participating in a search process. As expected, while SiMPSON outperforms SimPeer-like in search cost, it incurs higher index costs than SimPeer-like does. However, as in Figure 4(c), which shows the ratio between SiMPSON and SimPeer-like in terms of the index costs and between SimPeer-like and SiMPSON in terms of search costs, while SiMPSON uses as many as two times the number of indices created by SimPeer-like, SiMPSON is four times better than SimPeer-like in Covtype data set and seven times better than SimPeer-like in VEC and Color Histograms data sets in terms of search cost.

5.2 Effects of Dimensionality

In this experiment, we evaluate the effects of varying dimensionality. The result in Figure 4(d) shows that with the increase of dimensionality the index cost of SiMPSON slightly increases while that of SimPeer-like does not increase. On the other hand, the

**Fig. 5.** Effects of range query size

search cost of SimPeer-like increases very fast compared to that of SiMPSON, as shown in Figure 4(e). In general, even though we trade index cost for a better search cost, the result in Figure 4(f), which displays the ratio between SiMPSON and SimPeer-like in index costs and between SimPeer-like and SiMPSON in search costs, shows that our method gains more benefit in high dimensions. In particular, in a 32 dimensional space, while our method incurs an index cost as many as three times as that of SimPeer-like, it significantly reduces the search cost up to 40 times compared to SimPeer-like.

5.3 Effects of Range Query Size

In this experiment, we evaluate the effect of varying the radius of the range queries. Figure 5(a) shows the average number of nodes participating in a search process and Figure 5(b) shows the ratio between SimPeer-like and SiMPSON in search costs. The results show that in all datasets, SiMPSON is much better than SimPeer-like when the range query size is small and with the increase of query size from 0.01 to 0.04, the ratio between SimPeer-like and SiMPSON in search cost gradually reduces. This is because SiMPSON's search cost depends on the range query radius size while the search cost of SimPeer-like does not. By indexing only on max values of clusters, SimPeer-like always needs to search from the closest points of the query to the boundary of the intersecting partitions, and hence its search cost is less sensitive to the query size. However, when the query radius size increases from 0.04 to 0.05, the ratio between SimPeer-like and SiMPSON in search costs increases again. It is because with a bigger radius size, the search region of SimPeer-like may intersect more partitions, and hence incur more cost.

5.4 Performance of k-NN Query

In this experiment, we evaluate the performance of three methods used to estimate the initial radius of kNN queries as discussed in Section 4.2. The results are displayed in Figure 6 in which Method-B, Method-LO and Method-GO respectively denote the basic estimation method, the Locally Optimized method and the Globally Optimized method. The results show that in all data sets, Method-B is the worst while Method-LO takes the lead. The reason for which Method-GO incurs a higher cost than Method-LO is because in Method-GO, while trying to minimize the search radius, the system may under-estimate it, and hence there are not enough results returned in the first step. As a result, the system needs to increase the search radius to a safe value in the second step. This incurs a high cost. Nevertheless, Method-GO is still much better than the

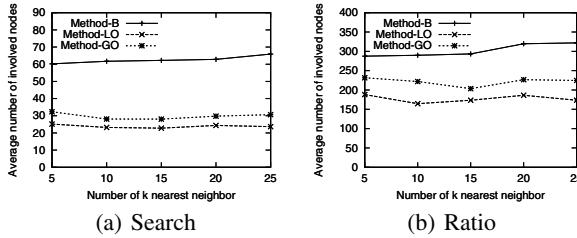


Fig. 6. A comparison of three k-NN query processing methods

basic method (it reduces approximately 50% and 25% of the search cost incurred by Method-B in Covtype and VEC data sets).

6 Conclusion

In this work we have introduced SiMPSON: a peer-to-peer system supporting similarity search in metric spaces. SiMPSON fits into the select category of P2P systems working natively in the more general environment of metric spaces, as opposed to the majority of systems that work natively in 1 or n -dimensional spaces. In its class, it differentiates itself by the use of a structured underlying architecture, taking advantage of all the benefits that come with such a structure. Compared with the available state-of-the-art in P2P similarity search, our method significantly improves the query retrieval process, while only slightly increasing the number of indexing steps. Extensive experimental results show that this improvement, for both range and kNN queries, increases directly proportional with the size of the network, adding ground to our scalability claims.

References

1. Banaei-Kashani, F., Shahabi, C.: Swam: a family of access methods for similarity-search in peer-to-peer data networks. In: ACM CIKM, pp. 304–313 (2004)
2. Bawa, M., Condé, T., Ganesan, P.: LSH forest: self-tuning indexes for similarity search. In: WWW, pp. 651–660 (2005)
3. Beckmann, N., Kriegel, H.-P., Schneider, R., Seeger, B.: The R*-tree: An efficient and robust access method for points and rectangles. In: ACM SIGMOD, pp. 322–331 (1990)
4. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Communications of the ACM 18(9), 509–517 (1975)
5. Blackard, J.A.: Covtype Data Set, Colorado State University (1998), <http://archive.ics.uci.edu/ml/datasets/Covtype>
6. Chavez, E., Navarro, G., Baeza-Yates, R., Marroquin, J.L.: Searching in metric spaces. ACM Computing Surveys 33(3), 273–321 (2001)
7. Chun, B., Culler, D., Roscoe, T., Bavier, A., Peterson, L., Wawrzoniak, M., Bowman, M.: Planetlab: An overlay testbed for broad-coverage services. ACM SIGCOMM Computer Communication Review 33(3) (2003)
8. Ciaccia, P., Patella, M., Zezula, P.: M-tree: An efficient access method for similarity search in metric spaces. In: VLDB, pp. 426–435 (1997)
9. Doulkeridis, C., Vlachou, A., Kotidis, Y., Vazirgiannis, M.: Peer-to-peer similarity search in metric spaces. In: VLDB (2007)

10. Falchi, F., Gennaro, C., Zezula, P.: A content-addressable network for similarity search in metric spaces. In: Moro, G., Bergamaschi, S., Joseph, S., Morin, J.-H., Ouksel, A.M. (eds.) DBISP2P 2005 and DBISP2P 2006. LNCS, vol. 4125, pp. 98–110. Springer, Heidelberg (2007)
11. Guttman, A.: R-trees: A dynamic index structure for spatial searching. In: ACM SIGMOD, pp. 47–57 (1984)
12. Hjaltason, G.R., Samet, H.: Index-driven similarity search in metric spaces. ACM Transactions on Database Systems (TODS) 28(4), 517–580 (2003)
13. Jagadish, H.V., Ooi, B.C., Tan, K.-L., Yu, C., Zhang, R.: iDistance: An adaptive B^+ -tree based indexing method for nearest neighbor search. ACM Transactions on Database Systems (TODS) 30(2), 364–397 (2005)
14. Jagadish, H.V., Ooi, B.C., Vu, Q.H.: BATON: A balanced tree structure for Peer-to-Peer networks. In: VLDB (2005)
15. Jagadish, H.V., Ooi, B.C., Vu, Q.H., Zhang, R., Zhou, A.: VBI-tree: a peer-to-peer framework for supporting multi-dimensional indexing schemes. In: ICDE (2006)
16. Karger, D., Kaashoek, F., Stoica, I., Morris, R., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: ACM SIGCOMM, pp. 149–160 (2001)
17. Li, M., Lee, W.-C., Sivasubramaniam, A.: DPTree: A balanced tree based indexing framework for peer-to-peer systems. In: ICNP (2006)
18. Novak, D., Zezula, P.: M-chord: a scalable distributed similarity search structure. In: InfoScale (2006)
19. Ooi, B.C., Tan, K.-L., Yu, C., Bressan, S.: Indexing the edges: a simple and yet efficient approach to high-dimensional indexing. In: ACM PODS (2000)
20. Ortega-Binderberger, M.: Image features extracted from a Corel image collection (1999), <http://kdd.ics.uci.edu/databases/CorelFeatures/CorelFeatures.data.html>
21. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S.: A scalable content-addressable network. In: ACM SIGCOMM, pp. 161–172 (2001)
22. Sellis, T., Roussopoulos, N., Faloutsos, C.: The R^+ -tree: A dynamic index for multi-dimensional objects. In: VLDB, pp. 507–518 (1987)

Uniform Sampling for Directed P2P Networks

Cyrus Hall and Antonio Carzaniga

Faculty of Informatics
University of Lugano
Lugano, Switzerland

hallc@lu.unisi.ch, antonio.carzaniga@unisi.ch

Abstract. Selecting a random peer with uniform probability across a peer-to-peer (P2P) network is a fundamental function for unstructured search, data replication, and monitoring algorithms. Such uniform sampling is supported by several techniques. However, current techniques suffer from sample bias and limited applicability. In this paper, we present a sampling algorithm that achieves a desired uniformity while making essentially no assumptions about the underlying P2P network. This algorithm, called *doubly stochastic converge* (DSC), iteratively adjusts the probabilities of crossing each link in the network during a random walk, such that the resulting transition matrix is doubly stochastic. DSC is fully decentralized and is designed to work on both directed and undirected topologies, making it suitable for virtually any P2P network. Our simulations show that DSC converges quickly on a wide variety of topologies, and that the random walks needed for sampling are short for most topologies. In simulation studies with FreePastry, we show that DSC is resilient to high levels of churn, while incurring a minimal sample bias.

1 Introduction

Our overarching goal is to create a generic “plug-and-play” framework for sampling properties (bandwidth, load, etc.) of interconnected peers in an arbitrary P2P network. Ideally, the resulting measurements should give an unbiased view of the current distribution of the properties over the network, which is useful for immediate parameter tuning as well as for a correct understanding of network dynamics over multiple sample runs.

In this paper we focus on a fundamental component of such a framework. Specifically, we implement a *uniform sampling function* that returns a peer chosen uniformly at random among all peers in a network. This function is applicable to networks of any topology, and requires no global knowledge. In addition to serving as a basis for monitoring, uniform random sampling is a useful building block in distributed systems in general, where it is used to support search, maintenance, replication, and load-balancing [12].

Existing techniques for uniform sampling are based on biased random walks or gossiping. In the first case, a node is selected at the end of a sufficiently long random walk in which the probabilities of following each link are adjusted to obtain a uniform visitation distribution across the network. Existing algorithms,

such as Metropolis-Hastings and maximum-degree, use local properties of nodes to compute their visitation probabilities, which they then use to bias the transition probabilities [23][4]. The locality of this information makes these algorithms practical and efficient. However, these algorithms also assume that all links are bidirectional, and therefore may not be universally applicable. In gossip-based sampling, nodes maintain a pool of addresses of other peers that is frequently exchanged and shuffled with other peers through a gossip protocol [5]. Gossip sampling is tunable and efficient in terms of traffic, and is effective for applications such as search, load-balancing, and topology construction. However, it is less appropriate for statistical sampling, especially with a bursty or high demand, because of the overhead of an increased gossip rate, which is necessary to provide independent and identically distributed samples (see Section 3).

Our contribution is to remove the assumption of bidirectional links in the network while maintaining desirable statistical properties (i.e., uniformity) for the sampling service. Non-bidirectional networks come in two types: networks with truly asymmetric links, and networks lacking a consistency protocol for tracking incoming links (e.g., DHTs). The algorithm we propose is fully distributed and uses only localized information. These features make it applicable to virtually any P2P system, as well as to a wide range of applications.

Our algorithm falls in the general category of random-walk sampling. The main idea is to avoid the calculation of each node's visitation probability, and instead to adjust the transition probabilities iteratively, converging to a state in which the sum of transition probabilities into each node equals 1. The resulting transition matrix is said to be *doubly stochastic*, and induces uniform visitation probabilities. In practice, the algorithm performs well on both static and dynamic topologies, keeping the ratio between the maximum and minimum sample probability below 1.2 for realistic churn conditions. Further, our algorithm generates link-biases that keep the expected sample walk length reasonably short, between 20 and 50 hops for 1000-node static (no churn) topologies of various kinds, and around 23 hops for 1000-node Pastry networks under churn.

In Section 2 we cover the relevant background necessary to explain our algorithm. Section 3 reviews previous work on P2P sampling. Section 4 presents a basic version of our *doubly stochastic converge* (DSC) algorithm, sketches a proof of its convergence, and presents a more advanced variant that reduces the length of the random walks and deals with failure. Section 5 evaluates DSC in simulations on static topologies, and within a concrete system under churn. Finally, Section 6 concludes with a discussion of future work.

2 Background Theory

We model a P2P network as a directed graph $G = (V, E)$ on $n = |V|$ vertices. For each vertex v , $N^-(v) = \{u \in V \setminus \{v\} : (u, v) \in E\}$ is the *in-neighborhood* of v , and $N^+(v) = \{u \in V \setminus \{v\} : (v, u) \in E\}$ is the *out-neighborhood* of v . Notice that N^- and N^+ do not contain v itself. For each pair of vertices u and v , p_{uv} is the *transition probability* from u to v . That is, p_{uv} is the probability of

proceeding from u to v in a random walk. Transition probabilities are such that $p_{uv} = 0$ when $(u, v) \notin E$, and $\sum_{v \in V} p_{uv} = 1$ for every vertex u , and therefore form a stochastic $n \times n$ matrix \mathbf{P} .

We model a random walk across the P2P network as a Markov chain with transition matrix \mathbf{P} . The vector \mathbf{x} denotes the probabilities of being at each vertex in the graph, and therefore in each state of the Markov chain. Given this distribution of probabilities \mathbf{x}_t at time t , one step in the random walk produces the distribution $\mathbf{x}_{t+1} = \mathbf{x}_t \mathbf{P}$. If the Markov chain is ergodic, then the probability to arrive at a given node after a sufficiently long walk stabilizes, independently of the starting point, to a unique *stationary distribution* π , where each node u has a *visitation probability* of $\pi(u)$. The *mixing-time* of \mathbf{P} is the minimal length of a walk that achieves a desired deviation from the stationary distribution.

A Markov chain is ergodic if every state (i.e., every peer) is reachable from any other state, and, beyond a certain path length, is reachable at all greater path lengths. Formally, there exists a number of steps q for which $\forall n \geq q : \mathbf{P}^n(u, v) > 0$ for all nodes u and v . In practice, a P2P network that is both strongly connected and aperiodic—reasonable assumptions—will have an ergodic Markov chain.

Since we want to sample peers uniformly starting from any node, our goal is to obtain a *uniform* stationary distribution. The theory of Markov chains provides a sufficient condition that leads to such a stationary distribution. Specifically, every ergodic Markov chain with a *doubly-stochastic* transition matrix has a uniform stationary distribution. A matrix is doubly-stochastic when every row and every column sums to 1. In other words, if the sum of the transition probabilities of all the *incoming* edges of every node is 1 (the sum of the transition probabilities of all the *outgoing* edges is 1 by definition) then the transition matrix is doubly-stochastic, and the stationary distribution is uniform.

3 Related Work

For the purpose of this paper, it is useful to distinguish two broad categories of techniques for uniform peer sampling over P2P systems. First we discuss techniques developed for specific P2P systems or topologies. Then we review more general techniques, which we further classify in two main subcategories: those based on random walks, and those based on gossip protocols.

King and Saia present a method to select a peer uniformly at random from any DHT that supports the common ID-based lookup function, where $get(x)$ returns the closest peer to ID x [6]. Similarly, studies of churn dynamics in DHTs often assume that peers can be sampled by looking up randomly selected values in the address space [7]. However, many DHTs show a preference toward storing long-lived nodes in routing tables in order to increase reliability, which biases the selection toward such nodes. Further, natural differences in the distribution of IDs biases toward peers that are responsible for larger portions of the ID-space. In general, any technique which samples peers in a manner correlated with session length will lead to biased results [28]. Beyond their statistical properties, these techniques are also limited in that they are only applicable to DHTs.

Moving away from system-specific techniques, one general way to sample peers at random is to use random walks, where a peer is sampled at the end of a sufficiently long random walk. However, since the probability of reaching each peer depends on the topology, and is generally not uniform, the walk must be biased in such a way as to obtain a uniform visitation probability. This is typically done in two steps: first, each node u computes for each neighbor v the ratio between its unbiased visitation probability and that of v . This is easily accomplished assuming that all links are bidirectional, as the ratio $\pi(u)/\pi(v)$ equals the ratio between the degrees of u and v , and can therefore be computed locally [2]. In the second step, an algorithm such as Metropolis-Hastings or maximum-degree [23,49] is used to bias the transition probability of each link in order to obtain a uniform stationary distribution π .

In practice, existing random-walk techniques are effective. However, as they assume that all links are bidirectional, they may not be universally applicable. Furthermore, the use of bidirectional connections may incur additional costs, such as NAT traversal, or additional state tracking for incoming links.

Other general-purpose sampling techniques are based on gossip protocols [5], which have also been shown to be useful in topology construction and maintenance [10], and in the estimation of global properties [11]. Gossip-based sampling amounts to picking an address from a local pool of peer addresses that is filled and continually shuffled through periodic “gossip” exchanges with other peers. The main advantage of gossip techniques is that extracting one sample is an immediate and local operation. Also, gossip protocols can be tuned to function at a low overhead, as only a few gossip messages per peer can guarantee a good level of randomness in the peer pools. However, these advantages deteriorate in the presence of frequent sample requests, as multiple requests can no longer return independent and identically distributed samples. There are two ways to improve the statistical quality of multiple samples in such scenarios: (1) increase the size of the peer pools, and (2) increase the gossip frequency. Unfortunately both actions also increase the overhead of the gossip protocol, and do so uniformly across the entire network. We also note that the cost of gossip messages must be added to the communication costs of the P2P system being sampled, even in the absence of churn, as gossip messages are exchanged with peers selected from the gossip peer pool, and not among the neighbors in the sampled P2P network.

4 The DSC Algorithm

One can uniformly sample a network by first assigning transition probabilities to edges such that the stationary distribution is uniform, and then performing random walks of sufficient lengths, bounded by the mixing-time. In this section we discuss an algorithm that solves the first of these requirements.

4.1 Basic DSC

As stated in Section 2, \mathbf{P} is stochastic by definition, as the outgoing edges of every node have a total probability of 1. Our task is to assign outgoing probabilities

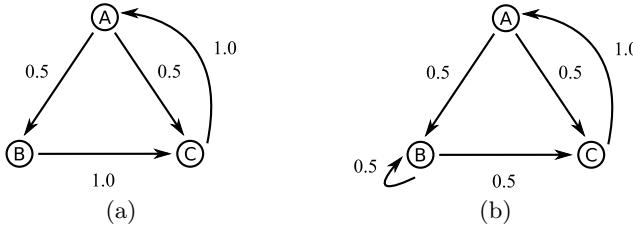


Fig. 1. An ergodic graph (a) Labeled with initial transition probabilities. (b) Balanced using a self-loop.

so that, for each node, the probabilities of the incoming edges also add up to 1, thereby making \mathbf{P} doubly stochastic. We refer to this process as *balancing*.

In a fully decentralized balancing algorithm, a node v can not directly control the probabilities assigned to its incoming edges, which are controlled by v 's predecessors. In fact, v may only know about a subset (possibly none) of its predecessors. Even if v could control the probability of some of its incoming edges, changing one of these probabilities would affect the balance of probabilities for other nodes. To gain some degrees of freedom for the purpose of balancing, we introduce an extra edge, the *self-loop*, linking each node back to itself. The self-loop is both an incoming and outgoing edge, and therefore can be used by a node to make up deficits in its own incoming probability.

Figure 1 exemplifies the difficulties of balancing, and the use of the self-loop. Neither node B or C can be balanced without removing edge (A, C) . B has a deficit of incoming probability, yet A cannot rebalance its out-probability to satisfy B without removing all probability from (A, C) . Conversely, C has too much incoming probability, and B can not redirect the excess probability elsewhere. Setting $p_{AC} = 0$ balances all in- and out-probabilities, but renders the graph periodic, and hence no longer ergodic. The solution, shown in Figure 1(b), is to use the self-loop (B, B) to increase the in-probability of B to 1, which reduces the in-probability of C to 1, balances \mathbf{P} , and keeps G ergodic. This leads directly to the core intuition of DSC: *increasing* the self-loop for nodes with in-probability deficits, and therefore reducing the probability across their outgoing edges, *decreases* the excess in-probability of other nodes.

More formally, we define the sum of in- and out-probability at a node v as $In(v) \triangleq p_{vv} + \sum_{u \in N^-(v)} p_{uv}$ and $Out(v) \triangleq p_{vv} + \sum_{u \in N^+(v)} p_{vu}$, where p_{vv} is the self loop. Both $In(v)$ and $Out(v)$ must sum to 1 in order for \mathbf{P} to be doubly stochastic. Clearly, increasing or decreasing the self-loop forces a decrease or increase, respectively, of the sum of probability across both $N^-(v)$ and $N^+(v)$. At any given time, a node is in one of three states: it has an in-probability surplus, so it is in $V^+ \triangleq \{v \in V : In(v) > 1\}$; it has an in-probability deficit, so it is in $V^- \triangleq \{v \in V : In(v) < 1\}$; or it is balanced, in $V^= \triangleq \{v \in V : In(v) = 1\}$.

The mean node in-probability is 1, as the sum of all in-probabilities equals the sum of all out-probabilities, which equals n since $Out(v) = 1$ for all v . Therefore, the total surplus of in-probability in V^+ equals the total deficit of in-probability in V^- . It follows that if we move nodes from V^- to $V^=$, nodes in V^+ will

eventually be forced into $V^=$. Once all nodes are in $V^=$, \mathbf{P} becomes doubly stochastic. Fortunately, moving a node v from V^- to $V^=$ is simple: increase the self-loop p_{vv} by $1 - In(v)$, bringing $In(v)$ to 1. This is the basic strategy of DSC. In particular, every node executes two steps:

1. Every t seconds, v updates its self-loop p_{vv} . When $In(v) < 1$, the self-loop is increased by $1 - In(v)$. If $In(v) \geq 1$, no action is taken.
2. Every t/f seconds, v sends updates to all $u \in N^+(v)$, notifying them of their current transition probability, $(1 - p_{vv})/|N^+(v)|$. Successor nodes store this value, using it to calculate $In(v)$ in step 1.

When step 1 is executed we say there has been an *update*. The time t should be selected with bandwidth, network latency, and churn in mind (see Section 5). A short t leads to quicker convergence, but also increasing the chance to miss updates if latency is high. The frequency of updates f can be set to 1 if on-time delivery is assured, or higher in the presence of packet losses or high latency. Step 2 of the basic version of DSC evenly balances and transmits the outgoing probabilities. The uniformity of transition probabilities across outgoing links is non-optimal for most topologies. In Section 4.3 we discuss a variant of this assignment strategy which can assign outgoing probabilities in a more effective way.

In a technical report [2], we prove that, as long as G is ergodic, basic DSC iteratively updates \mathbf{P} so that, in the limit, \mathbf{P} converges to a doubly stochastic matrix. The main intuition behind the proof is that the ergodicity of the graph assures that any reduction of out-probability at a node v in V^- (caused by the node increasing its self-loop) will eventually reach a set of nodes in V^+ , the members of which will each absorb some fraction of the change, leading to a global reduction of in-probability deficit. A key point of our proof is that neither the propagation time (number of updates), nor the amount of in-probability absorbed by u , depend on time. As a result, the global in-probability deficit converges exponentially to zero, leading to a doubly-stochastic transition matrix.

Within our proof, we also establish a lower bound for the reduction factor of the overall deficit. However, this bound is very loose. Therefore, we do not use it to evaluate DSC, and instead rely on simulation (see Section 5). In practice, DSC is quite fast in converging toward a doubly-stochastic bias, although various factors can influence its rate of convergence. One such factor is the ordering of node updates. For example, in Figure 1, updating A and C first is clearly not optimal. We have observed that different update ordering may change the rate of convergence by a factor of 2 for large topologies.

4.2 Failure and Relaxation

Churn events (i.e., nodes joining or leaving the network) have the effect of increasing self-loops. A node joining the network may move its successors from $V^=$ or V^- into V^+ . Such successors may find themselves in the peculiar state of having a surplus of in-probability *and* a self-loop greater than 0. This is never the case in the absence of churn. Nodes leaving the network have a similar effect.

Predecessors of a leaving node u will increase their probabilities across remaining out-edges, while successors will have to increase their self-loop to make up for lost in-probability. Just as when a node joins, a leaving node u can force some $v \in N^+(q) : q \in N^-(u)$ back into V^+ with a non-zero self-loop.

Taken together, these two phenomena lead to ever increasing self-loops, which is problematic as high self-loops tend to increase the mixing time. If we intend to keep a reasonable mixing time in a network with churn, we cannot rely on an algorithm that only increases self-loops. We therefore allow DSC to also lower self-loops. In particular, a node $v \in V^+$ with a self loop $p_{vv} > 0$ decreases its self loop by $\min(p_{vv}, 1 - In(v))$, setting it to zero, or as close to zero as possible. In the same operation, v must also raise the transition probabilities to its successors accordingly. In turn, this can cause $q \in N^+(u)$ to also lower their self-loop, and so on. However, the effect is dampened each successive step away from v , and is unlikely to propagate to the entire network, as small increases in probability are absorbed by nodes in V^- or $V^=$ with non-zero self-loops.

Relaxation is essential to the functioning of DSC in a network with churn. Without it, self-loops will approach 1, leading to higher and higher sample walk lengths. Importantly, relaxation does not require a channel of communication in the opposite direction of the network links, allowing it to work with basic DSC.

4.3 Feedback

Basic DSC does not assume bidirectional connectivity, and therefore does not use information about the state of a node's successors. In particular, basic DSC spreads the outgoing probabilities uniformly across successors, without considering their individual state. This can slow convergence and, more seriously, increase the mixing-time due to high self-loops probabilities. Whenever bidirectional links exist, feedback from successors can be used to weight outgoing probabilities. Specifically, in addition to storing the transition probability sent by a predecessor v , a node u may reply with an acknowledgment containing $In(u)$. v then uses this value to first bias p_{vu} , and then to renormalize all its other out-probabilities:

1. $p_{vu} \leftarrow p_{vu} / In(u)$ (bias)
2. $\forall q \in N^+(v) : (p_{vq} \leftarrow (p_{vq} / \sum_{q \in N^+(v)} p_{vq}) (1 - p_{vv}))$ (normalize)

The weighting is only applied once every t seconds, no matter how high f may be, so that lossy links are not inappropriately biased against.

5 Simulation Analysis

Our analysis of DSC considers two main questions: (1) How does DSC perform across a wide range of topologies, both in terms of convergence speed, and expected walk length? (2) Is DSC resilient under churn conditions? To characterize convergence and walk lengths, we use a discrete-event simulation of DSC. To analyze the effects of churn, we run a concrete implementation of DSC integrated within the FreePastry framework¹ under various levels of churn. A more extensive analysis of these experiments is available in a technical report [12].

¹ <http://freepastry.rice.edu>; an open-source implementation of Pastry [13].

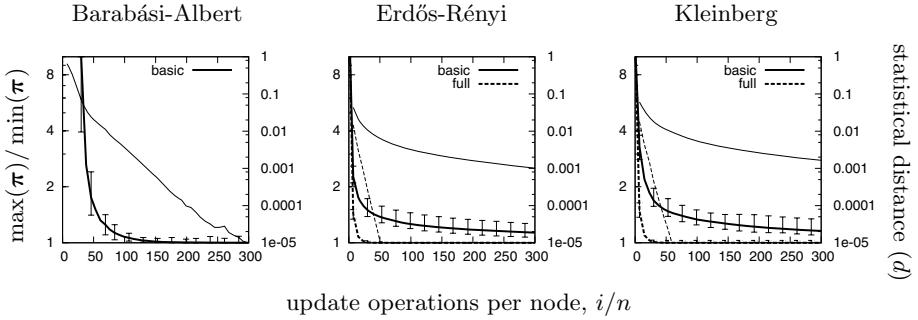


Fig. 2. Uniformity ratio r (thick) and statistical distance d (thin) after i/n updates

We simulate DSC over a diverse set of generated graphs. In particular, we use three types of generative models commonly used in P2P research:

- *Kleinberg*: nodes are connected in a grid, along with q long distance links chosen with probability inversely proportional to the squared distance [4].
- *Erdős-Rényi*: edges are selected independently with a fixed probability [5].
- *Barabási-Albert*: nodes are connected to high-degree nodes with preferential-attachment factor 0.5, creating a power-law degree distribution [5].

We use variants of these standard models to obtain directed graphs. All graphs have the same number of vertices $n = 1000$ and, with minor differences, the same number $|E| \approx n \log n$ of edges, which corresponds to a node–edge ratio found in many deployed systems. We run two versions of DSC: basic and full, where the latter includes feedback and relaxation. To measure the quality of the resulting sampling service, we compute two values, the ratio $r = \max(\pi)/\min(\pi)$, which highlights the worst-case difference in sample probability, and therefore gives a very conservative characterization of the uniformity achieved by DSC, and the statistical distance $d(\frac{1}{n}, \pi) = \frac{1}{2} \sum_{s \in S} |\frac{1}{n} - \pi(s)|$, which measures the normalized absolute difference between π and the uniform distribution.

Figure 2 shows the ratio r and statistical distance d as a function of the number of updates per peer, i/n , where i is the total number of updates over the network. Notice that updates execute in parallel, so i/n represents the number of rounds of update of the network as a whole. We simulate DSC on 50 topologies of each type, and plot the median value of r and d , with error bars for r showing the 10th and 90th percentile. In all situations DSC converges exponentially fast to the optimal ratio of $r = 1$. For Erdős-Rényi and Kleinberg topologies, full DSC performs better, reaching a tight ratio $r \leq 1.01$ in just 30 rounds. Reasonable values of d are achieved in less than 10 rounds for full DSC, and 50 rounds without feedback. Barabási-Albert topologies exhibit different behavior: basic DSC converges quickly, reaching $r = 1.01$ in 160 rounds, but full DSC experiences severe oscillations in r , and converges much more slowly. Further analysis reveals that this is due to in-probability oscillating between V^+ and V^- at low in-degree nodes, behavior which is caused by feedback. This problem could be eliminated by making feedback sensitive to successor in-degree.

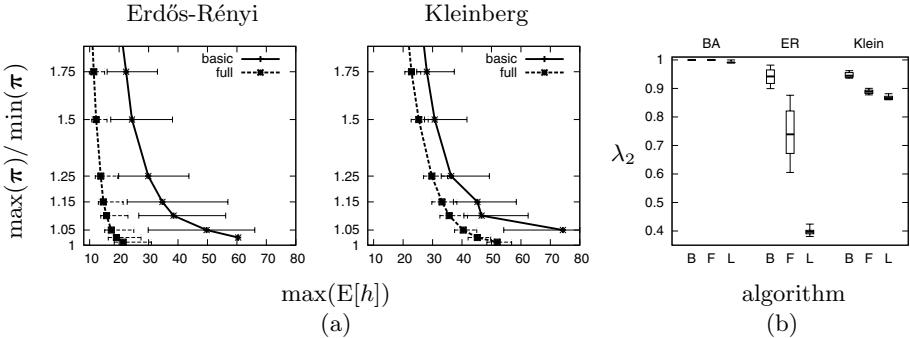


Fig. 3. (a) Relation between uniformity ratio r and maximum expected walk length. (b) Second eigenvalue for basic DSC (B), full DSC (F), and linear-programming (L), for each topology type: Barabási-Albert (BA), Erdős-Rényi (ER), and Kleinberg (Klein).

Quick convergence indicates that the cost of balancing is low, but does not say anything about the costs of sampling. This latter cost is determined by the mixing-time, that is, the number of hops after which a random walk becomes independent of its starting node. We analyze this cost using basic results from Markov-chain theory [12]. After the convergence of each topology, we calculate the maximum expected walk-length $\max(E[h])$ for different target values of r .

Figure 3(a) shows the results of this analysis, graphing the median value, with error bars representing the 10th and 90th percentiles. Reasonable values of r are achievable in less than 25 hops for Erdős-Rényi and 45 hops for Kleinberg with full DSC. However, feedback in full DSC can not significantly improve walk-lengths for Kleinberg topologies. This is because feedback takes advantage of irregularity in link structure, and Kleinberg topologies are highly regular. Barabási-Albert topologies are not shown, as $\max(E[h])$ can be as high as 10^7 for $r = 1.05$. All of our Barabási-Albert topologies have several extremely weakly connected nodes with very low initial visitation probability (typically less than 10^{-6}) which develop extremely high self-loops (often greater than 0.995).

In evaluating walk lengths in DSC, we would like to compare them with the best achievable walk lengths for each topology. Unfortunately, we are not aware of a suitable optimization algorithm for topologies of the type and scale we considered. In order to obtain an estimate of the minimal walk length, we try to balance the transition probabilities using a linear-programming optimization. In particular, we use an objective function that minimizes self-loops while trying to evenly balance out-probability [12]. We then compare DSC with the linear-programming solution using the second-largest eigenvalue modulus λ_2 of the transition matrix \mathbf{P} (lower values of λ_2 indicate better walk lengths [16]).

The results, shown in Figure 3(b), demonstrate that there is room for improvement in the way DSC assigns probability across edges. The linear-programming method minimizes λ_2 to a greater extent thanks to its global view of the network, whereas DSC operates in a completely decentralized manner. We are considering ways to improve DSC's weightings by integrating more information into

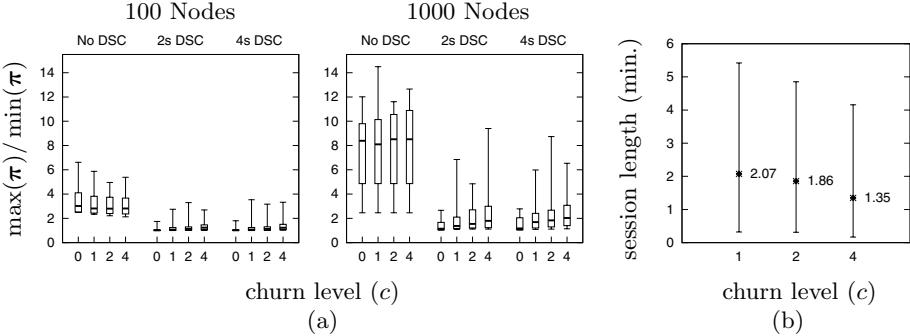


Fig. 4. (a) Uniformity ratio r under churn. (b) Session lengths.

feedback. The end goal is to move transition probability to those successors with low in-probability [3].

In the second part of our evaluation, we study the behavior of DSC under churn by running it within FreePastry, an implementation of the fault tolerant, locality-aware Pastry DHT [13]. Our churn model follows the findings of Stutzbach and Rejaie and their study of a DHT based on Kademlia [8,12]. We use FreePastry in simulation mode over 100 and 1000 nodes, with the churn process starting as soon as the last node has joined. After the initial join period is complete, we measure r for 10 minutes.

The graphs in Figure 4(a) show the ratio r (median, min, max, 10th, and 90th percentile) for different levels of churn. $c = 1$ indicates churn dynamics very similar to those measured by Stutzbach and Rejaie, while $c = 2, 4$ means twice and four times those levels [8,12]. Figure 4(b) shows the session lengths (median, first, and third quartile) for each value of c . We first measure the ratio r without DSC, then with DSC with updates every 2 and 4 seconds ($t = 2$ or 4). The results show that DSC achieves high uniformity of sampling, keeping the ratio r under 2 for more than 50% of the topologies under all churn levels, and does even better at $c = 1$.

We study the mixing-times in the Pastry experiments with the same methodology used for the static topologies.

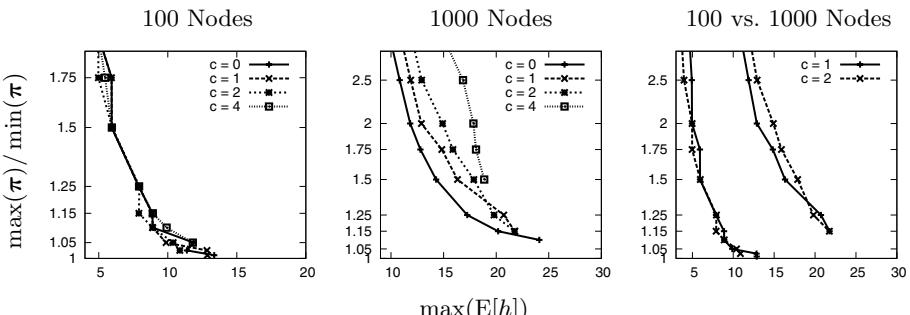


Fig. 5. Uniformity ratio r and maximum expected walk length under churn

Figure 5 displays the relation between the ratio r and the walk lengths under varying levels of churn, and shows that DSC achieves good uniformity with short sampling walks. In fact, 1000-node Pastry networks, with or without churn, have shorter or comparable walk lengths than any of the static topologies we studied. Furthermore, the graph comparing the 100 and 1000-node topologies suggests a sub-linear growth of the length of the walks as a function of the size of the network.

6 Conclusion

We have presented DSC, a distributed algorithm that balances the transition probabilities of peers in a directed P2P topology such that random walks of sufficient can uniformly sample the network. We gave a proof sketch of convergence for DSC, and showed, through simulations, that the rate of convergence is usable in many concrete scenarios, and remains so under realistic levels of churn. Further, we found the sample-length of DSC-biased topologies to be acceptable, and that churn has minimal effects on sample probability.

Active development continues on DSC. First, we are continuing to simulate DSC on networks of increasing sizes to study its scaling properties. In particular, we want to determine how scalable relaxation is for different topology types. Further, we would like to better understand the dynamics of both the stationary distribution and random walks under churn. Second, DSC has been designed to function in an opportunistic manner, so as to incur little or no traffic overhead, by piggy-backing on existing P2P traffic. We are working to integrate such piggy-backing with FreePastry using an aggregation technique we are developing. Finally, to make DSC usable in a concrete deployment, we are evaluating several efficient algorithms to estimate the minimal necessary walk lengths.

We would also like to return to a more theoretical analysis of DSC, first considering the effects of feedback and relaxation on convergence, and then trying to tighten the bounds on convergence. We are also very interested in trying to improve the biasing. Compared to approximate linear programming solutions, DSC biasing produces longer walks, and a method similar to the one presented by Awan et. al could be appropriate [3]. Another option to is to explore distributed approximations of various iterative minimizations of the mixing time [17].

Acknowledgments. This research was supported in part by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS) under Swiss National Science Foundation grant number 5005-67322. The authors are grateful to Shane Legg, Mark Carman, Thomas Shrimpton, and the anonymous reviewers for their useful comments and suggestions.

References

1. Gkantsidis, C., Mihail, M., Saberi, A.: Random walks in peer-to-peer networks. In: Proceedings of the 23rd Conference of the IEEE Communications Society, INFOCOM (2004)
2. Stutzbach, D., Rejaie, R., Duffield, N., Sen, S., Willinger, W.: On unbiased sampling for unstructured peer-to-peer networks. In: Proceedings of the 6th Internet Measurement Conference, IMC (2006)

3. Awan, A., Ferreira, R.A., Jagannathan, S., Grama, A.: Distributed uniform sampling in unstructured peer-to-peer networks. In: Proceedings of the 39th Annual Hawaii International Conference on System Sciences, HICSS (2006)
4. Datta, S., Kargupta, H.: Uniform data sampling from a peer-to-peer network. In: Proceedings of the 27th International Conference on Distributed Computing Systems, ICDCS (2007)
5. Jelasity, M., Voulgaris, S., Guerraoui, R., Kermarrec, A.M., van Steen, M.: Gossip-based peer sampling. *ACM Transactions on Computing Systems* 25(3) (2007)
6. King, V., Saia, J.: Choosing a random peer. In: Proceedings of the 23rd Symposium on Principles of Distributed Computing, PODC (2004)
7. Bhagwan, R., Savage, S., Voelker, G.: Understanding availability. In: Proceedings of the 2nd International Workshop on Peer-To-Peer Systems, IPTPS (2003)
8. Stutzbach, D., Rejaie, R.: Understanding churn in peer-to-peer networks. In: Proceedings of the 6th Internet Measurement Conference, IMC (2006)
9. Zhong, M., Shen, K., Seiferas, J.: The convergence-guaranteed random walk and its applications in peer-to-peer networks. *ACM Transactions on Computers* 57(5) (2008)
10. Jelasity, M., Babaoglu, O.: T-man: Gossip-based overlay topology management. In: Brueckner, S.A., Di Marzo Serugendo, G., Hales, D., Zambonelli, F. (eds.) ESOA 2005. LNCS, vol. 3910, pp. 1–15. Springer, Heidelberg (2006)
11. Kempe, D., Dobra, A., Gehrke, J.: Gossip-based computation of aggregate information. In: Proceedings of the 44th Symposium on Foundations of Computer Science, FOCS (2003)
12. Hall, C., Carzaniga, A.: Doubly stochastic converge: Uniform sampling for directed P2P networks. Technical report, University of Lugano, Lugano, Switzerland (2009)
13. Rowstron, A., Druschel, P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Guerraoui, R. (ed.) Middleware 2001. LNCS, vol. 2218, p. 329. Springer, Heidelberg (2001)
14. Kleinberg, J.: The small-world phenomenon: an algorithm perspective. In: Proceedings of the 32nd Symposium on Theory of Computing, STOC (2000)
15. Albert, R., Barabási, A.L.: Statistical mechanics of complex networks. *Reviews of Modern Physics* 74(1) (2002)
16. Sinclair, A.: Improved bounds for mixing rates of marked chains and multicommodity flow. In: Simon, I. (ed.) LATIN 1992. LNCS, vol. 583. Springer, Heidelberg (1992)
17. Boyd, S., Diaconis, P., Xiao, L.: Fastest mixing markov chain on a graph. *SIAM Review* 46(4) (2004)

Adaptive Peer Sampling with Newscast*

Norbert Tölgyesi¹ and Márk Jelasity²

¹ University of Szeged and Thot-Soft 2002 Kft., Hungary
ntolgyesi@gmail.com

² University of Szeged and Hungarian Academy of Sciences, Hungary
jelasity@inf.u-szeged.hu

Abstract. The peer sampling service is a middleware service that provides random samples from a large decentralized network to support gossip-based applications such as multicast, data aggregation and overlay topology management. Lightweight gossip-based implementations of the peer sampling service have been shown to provide good quality random sampling while also being extremely robust to many failure scenarios, including node churn and catastrophic failure. We identify two problems with these approaches. The first problem is related to message drop failures: if a node experiences a higher-than-average message drop rate then the probability of sampling this node in the network will decrease. The second problem is that the application layer at different nodes might request random samples at very different rates which can result in very poor random sampling especially at nodes with high request rates. We propose solutions for both problems. We focus on Newscast, a robust implementation of the peer sampling service. Our solution is based on simple extensions of the protocol and an adaptive self-control mechanism for its parameters, namely—without involving failure detectors—nodes passively monitor local protocol events using them as feedback for a local control loop for self-tuning the protocol parameters. The proposed solution is evaluated by simulation experiments.

1 Introduction

In large and dynamic networks many protocols and applications require that the participating nodes be able to obtain random samples from the entire network. Perhaps the best-known examples are gossip protocols [1], where nodes have to periodically exchange information with random peers. Other P2P protocols that also require random samples regularly include several approaches to aggregation [2][3] and creating overlay networks [4][5][6][7], to name just a few.

One possibility for obtaining random samples is to maintain a complete membership list at each node and draw samples from that list. However, in dynamic networks this approach is not feasible. Several approaches have been proposed to implementing peer sampling without complete membership information, for example, based on random walks [8] or gossip [9][10][11].

* M. Jelasity was supported by the Bolyai Scholarship of the Hungarian Academy of Sciences. This work was partially supported by the Future and Emerging Technologies programme FP7-COSI-ICT of the European Commission through project QLectives (grant no.: 231200).

Algorithm 1. Newscast

1: loop	12: procedure ONUPDATE(m)
2: wait(Δ)	13: buffer \leftarrow merge(view, {myDescriptor})
3: $p \leftarrow$ getRandomPeer()	14: send updateResponse(buffer) to m .sender
4: buffer \leftarrow merge(view, {myDescriptor})	15: buffer \leftarrow merge(view, m .buffer)
5: send update(buffer) to p	16: view \leftarrow selectView(buffer)
6: end loop	17: end procedure
7:	
8: procedure ONUPDATERESPONSE(m)	
9: buffer \leftarrow merge(view, m .buffer)	
10: view \leftarrow selectView(buffer)	
11: end procedure	

Gossip-based solutions are attractive due to their low overhead and extreme fault tolerance [12]. They tolerate severe failure scenarios such as partitioning, catastrophic node failures, churn, and so on. At the same time, they provide good quality random samples.

However, known gossip-based peer sampling protocols implicitly assume the *uniformity* of the environment, for example, message drop failure and the rate at which the application requests random samples are implicitly assumed to follow the same statistical model at each node. As we demonstrate in this paper, if these assumptions are violated, gossip based peer sampling can suffer serious performance degradation. Similar issues have been addressed in connection with aggregation [13].

Our contribution is that, besides drawing attention to these problems, we propose solutions that are based on the idea that system-wide adaptation can be implemented as an aggregate effect of simple adaptive behavior at the local level. The solutions for the two problems related to message drop failures and application load both involve a local control loop at the nodes. The local decisions are based on passively observing the local events and do *not* involve explicit failure detectors, or reliable measurements. This feature helps to preserve the key advantages of gossip protocols: simplicity and robustness.

2 Peer Sampling with Newscast

In this section we present a variant of gossip-based peer sampling called Newscast (see Algorithm 1). This protocol is an instance of the protocol scheme presented in [12], tuned for maximal self-healing capabilities in node failure scenarios. Here, for simplicity, we present the protocol without referring to the general scheme.

Our system model assumes a set of nodes that can send messages to each other. To send a message, a node only needs to know the address of the target node. Messages can be delayed by a limited amount of time or dropped. Each node has a partial view of the system (view for short) that contains a constant number of node descriptors. The maximal size of the view is denoted by c . A node descriptor contains a node address that can be used to send messages to the node, and a timestamp.

The basic idea is that all the nodes exchange their views periodically, and keep only the most up-to-date descriptors of the union of the two views locally. In addition, every time a node sends its view (update message) it also includes an up-to-date descriptor of itself.

Parameter Δ is the period of communication common to all nodes. Method `getRandomPeer` simply returns a random element from the current view. Method `merge` first merges the two lists it is given as parameters, keeping only the most up-to-date descriptor for each node. Method `selectView` selects the most up-to-date c descriptors.

Applications that run on a node can request random peer addresses from the entire network through an API that Newscast provides locally at that node. The key element of the API is method `getRandomPeer`. Though in a practical implementation this method is not necessarily identical to the method `getRandomPeer` that is used by Newscast internally (for example, a tabu list may be used), in this paper we assume that the application is simply given a random element from the current view.

Lastly, we note that although this simple version of Newscast assumes that the clocks of the nodes are synchronized, this requirement can easily be relaxed. For example, nodes could adjust timestamps based on exchanging current local times in the update messages, or using hop-count instead of timestamps (although the variant that uses hop-count is not completely identical to the timestamp-based variant).

3 Problem Statement

We identify two potential problems with Newscast noting that these problems are common to all gossip-based peer sampling implementations in [12]. The first problem is related to message drop failures, and the second is related to unbalanced application load.

3.1 Message Drop Failure

Gossip-based peer sampling protocols are designed to be implemented using lightweight UDP communication. They tolerate message drop well, as long as each node has the same failure model. This assumption, however, is unlikely to hold in general. For example, when a lot of failures occur due to overloaded routers that are close to a given node in the Internet topology, then UDP packet loss rate can be higher than average at the node. In this case, fewer copies of the descriptor of the node will be present in the views of other nodes in the network violating the uniform randomness requirement of peer sampling.

Figure 11 illustrates this problem in a network where there is no failure, only at a single node. This node experiences a varying drop rate, which is identical for both incoming and outgoing messages. The Figure shows the representation of the node in the network as a function of drop rate. Note the non-linearity of the relation. (For details on experimental methodology see Section 4.3)

3.2 Unbalanced Application Load

At different nodes, the application layer can request random samples at varying rates. Gossip-based peer sampling performs rather poorly if the required number of samples is much higher than the view size c over a time period of Δ as nodes participate in only two view exchanges on average during that time (one initiated and one passive). Figure 11 illustrates the problem. It shows the number of unique samples provided by

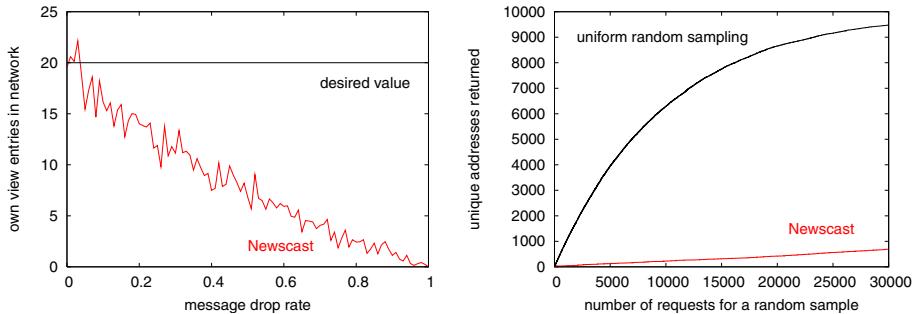


Fig. 1. Illustrations of two problems with gossip-based peer sampling. Parameters: $c = 20$, $N = 10,000$, the number of own entries in network is the average of 100 consecutive cycles

Newscast as a function of time at a node where the application requests random samples at a uniform rate with a period of $\Delta/1000$.

4 Message Drop Failure

4.1 Algorithm

We would like to adapt the period Δ of Newscast (Algorithm 1) at each node in such a way that the representation of each node in the network is identical, irrespective of message drop rates. This means that ideally all the nodes should have c copies of their descriptor since the nodes hold Nc entries altogether, where N is the number of nodes.

We make the assumption that for each overlay link (i, j) there is a constant drop rate $\lambda_{i,j}$, and all the messages passing from i to j are dropped with a probability of $\lambda_{i,j}$. According to [14] this is a reasonable assumption. For the purpose of performing simulation experiments we will introduce a more specific structural model in Section 4.2.

The basic idea is that all the nodes passively monitor local messages and based on this information they decide whether they are under- or overrepresented in the network. Depending on the result, they slightly decrease or increase their period in each cycle, within the interval $[\Delta_{min}, \Delta_{max}]$.

Let us now elaborate on the details. All the nodes collect statistics about the incoming and outgoing message types in a moving time window. The length of this time window is Δ_{stat} . The statistics of interest (along with notations) are the following (note that we have omitted the node id from the notations for the sake of clarity): the number of update messages sent (u_{out}), the number of update messages received (u_{in}), and the number of update response messages received (r_{in}).

From these statistics, a node approximates its representation in the network (n). Clearly, n is closely related to u_{in} . The exact relationship that holds for the expectation of u_{in} is

$$E(u_{in}) = \frac{1}{c} P_{in} n \Delta_{stat} \phi_{avg}, \quad (1)$$

where c is the view size, P_{in} is the probability that a message sent from a random node is received, and ϕ_{avg} is the average frequency of the nodes in the network at which they send update messages.

The values for this equation are known except P_{in} and ϕ_{avg} . Note however, that ϕ_{avg} is the same for all nodes. In addition, ϕ_{avg} is quite close to $1/\Delta_{max}$ if most of the nodes have close to zero drop rates, which is actually the case in real networks [14]. For these two reasons we shall assume from now on that $\phi_{avg} = 1/\Delta_{max}$. We validate this assumption by performing extensive experiments (see Section 4.3).

The only remaining value to approximate is P_{in} . Here we focus on the symmetric case, when $\lambda_{i,j} = \lambda_{j,i}$. It is easy to see that here $E(r_{in}) = P_{in}^2 u_{out}$, since all the links have the same drop rate in both directions, and to get a response, the update message first has to reach its destination and the response has to be delivered as well. This gives us the approximation $P_{in} \approx \sqrt{r_{in}/u_{out}}$.

Once we have an approximation for P_{in} and calculate n , each node can apply the following control rule after sending an update:

$$\Delta(t+1) = \begin{cases} \Delta(t) - \alpha + \beta & \text{if } n < c \\ \Delta(t) + \alpha + \beta & \text{if } n > 2c \\ \Delta(t) + \alpha(n/c - 1) + \beta & \text{otherwise,} \end{cases} \quad (2)$$

where we also bound Δ by the interval $[\Delta_{min}, \Delta_{max}]$. Parameters α and β are positive constants; α controls the maximal amount of change that is allowed in one step towards achieving the desired representation c , and β is a term that is included for stability: it always pulls the period towards Δ_{max} . This stabilization is necessary because otherwise the dynamics of the system will be scale invariant: without β , for a setting of periods where nodes have equal representation, they would also have an equal representation if we multiplied each period by an arbitrary factor. It is required that $\beta \ll \alpha$.

Although we do not evaluate the asymmetric message drop scenario in this paper (when $\lambda_{i,j} \neq \lambda_{j,i}$), we briefly outline a possible solution for this general case. As in the symmetric case, what we need is a method to approximate P_{in} . To achieve this we need to introduce an additional trick into Newscast: let the nodes send R independent copies of each update response message ($R > 1$). Only the first copy needs to be a full message, the remaining ones could be simple ping-like messages. In addition, the copies need not be sent at the same time, they can be sent over a period of time, for example, over one cycle. Based on these ping messages we can use the approximation $P_{in} \approx r_{in}/(Rn_{in})$, where n_{in} is the number of *different* nodes that sent an update response. In other words, we can directly approximate P_{in} by explicitly sampling the distribution of the incoming drop rate from random nodes.

This solution increases the number of messages sent by a factor of R . However, recalling that the message complexity of gossip-based peer sampling is approximately one UDP packet per node during any time period Δ , where Δ is typically around 10 seconds, this is still negligible compared to most current networking applications.

4.2 Message Drop Failure Model

In our model we need to capture the possibility that nodes may have different message drop rates, as discussed in Section 3.1. The structure of our failure model is illustrated in

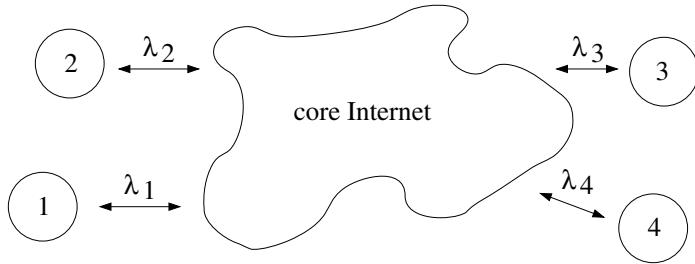


Fig. 2. The topological structure of the message drop failure model

Figure 2. The basic idea is that all the nodes have a link to the core Internet that captures their local environment. The core is assumed to have unbiased failure rates; that is, we assume that for any given two links, the path that crosses the core has an identical failure model. In other words, we simply assume that node-specific differences in failure rates are due to effects that are close to the node in the network. We define the drop rate of a link (i, j) by $\lambda_{i,j} = \lambda_i \lambda_j$.

We assume that each message is dropped with a probability independent of previous message drop events. This is a quite reasonable assumption as it is known that packet loss events have negligible autocorrelation if the time lag is over 1000 ms [14].

It should be mentioned that the algorithm we evaluate does *not* rely on this model for correctness. This model merely allows us (i) to control the failure rate at the node level instead of the link level and (ii) to work with a compact representation.

As for the actual distributions, we evaluate three different scenarios:

Single-Drop. As a baseline, we consider the model where any node i has $\lambda_i = 0$, except a single node that has a non-zero drop-rate.

Uniform. Drop rate λ_i is drawn from the interval $[0, 0.5]$ uniformly at random.

Exponential. Based on data presented in [14] we approximate $100\lambda_i$ (the drop rate expressed as the percentage of dropped messages) by an exponential distribution with parameter $1/4$, that is, $P(100\lambda_i < x) = 1 - e^{-0.25x}$. The average drop rate is thus 4%, and $P(100\lambda_i > 20\%) \approx 0.007$.

4.3 Evaluation

In order to carry out simulation experiments, we implemented the algorithm over the event-based engine of PeerSim [15]. The parameter space we explored contains every combination of the following settings: the drop rate distribution is single-drop, uniform, or exponential; α is $\Delta_{max}/1000$ or $\Delta_{max}/100$; β is 0, $\alpha/2$, or $\alpha/10$; and Δ_{stat} is Δ_{max} , $2\Delta_{max}$, $5\Delta_{max}$, or $20\Delta_{max}$. If not otherwise stated, we set a network size of $N = 5000$ and simulated no message delay. However, we explored the effects of message delay and network size over a subset of the parameter space, as we explain later. Finally, we set $c = 20$ and $\Delta_{min} = \Delta_{max}/10$ in all experiments.

The experiments were run for 5000 cycles (that is, for a period of $5000\Delta_{max}$ time units), with all the views initialized with random nodes. During a run, we observed the dynamic parameters of a subset of nodes with different failure rates. For a given failure

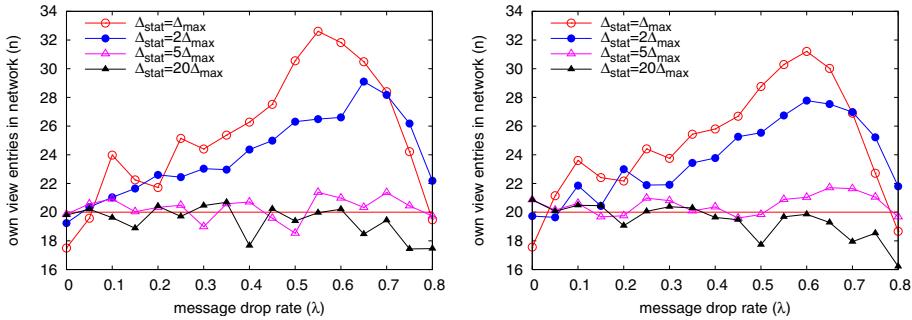


Fig. 3. The effect of Δ_{stat} on n . The fixed parameters are $N = 5,000$, $\alpha = \Delta_{max}/1000$, $\beta = \alpha/10$, exponential drop rate distribution. Message delay is zero (left) or uniform random from $[0, \Delta_{max}/10]$ (right).

rate, all the plots we present show average values at a single node of the given failure rate over the last 1500 cycles, except Figure 6 that illustrates the dynamics (convergence and variance) of these values.

Let us first consider the role of Δ_{stat} (see Figure 3). We see that small values introduce a bias towards nodes with high drop rates. The reason for this is that with a small window it often happens that no events are observed due to the high drop rate, which results in a maximal decrease in Δ in accordance with the control rule in (2). We fix the value of $20\Delta_{max}$ from now on.

In Figure 3 we also notice that the protocol tolerates delays very well, just like the original version of Newscast. For parameter settings that are not shown, delay has no noticeable effect either. This is due to the fact that we apply no failure detectors explicitly, but base our control rule just on passive observations of average event rates that are not affected by delay.

Figure 4 illustrates the effect of coefficients α and β . The strongest effect is that the highest value of β introduces a bias against nodes with high drop rates. This is because a high β strongly pushes the period towards its maximum value, while nodes with high drop rates need a short period to get enough representation.

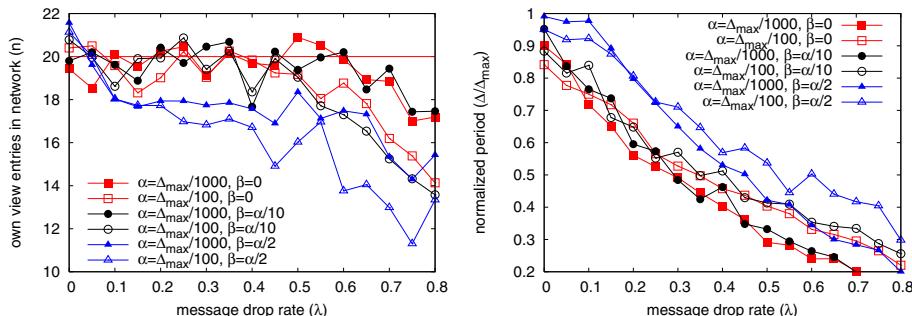


Fig. 4. The effect of α and β on n . The fixed parameters are $N = 5,000$, $\Delta_{stat} = 20\Delta_{max}$, exponential drop rate distribution.

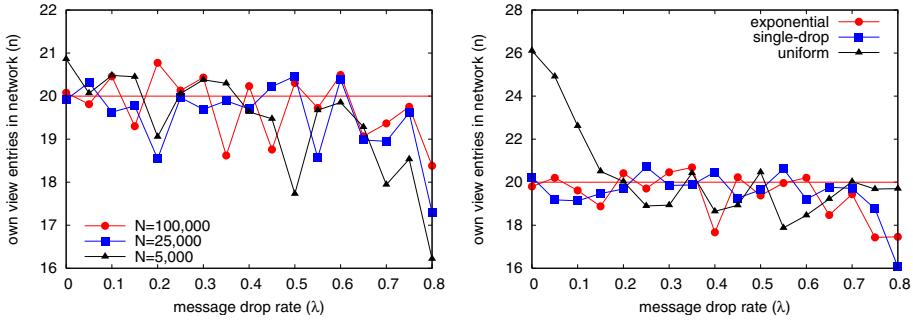


Fig. 5. The effect of network size and drop rate distribution on n . The fixed parameters are $N = 5,000$ (right), $\Delta_{stat} = 20\Delta_{max}$, $\alpha = \Delta_{max}/1000$, $\beta = \alpha/10$, exponential drop rate distribution (left)

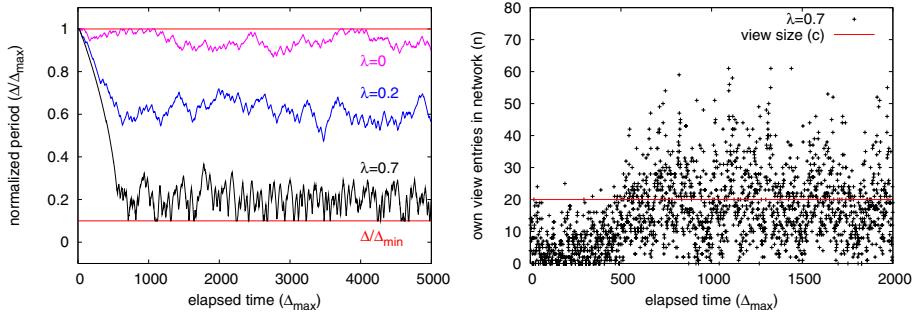


Fig. 6. The dynamics of Δ and n for $N = 5,000$, $\Delta_{stat} = 20\Delta_{max}$, $\alpha = \Delta_{max}/1000$, $\beta = \alpha/10$, exponential drop rate distribution

In general, the smaller value for α is more stable than the higher value for all values of β , which is not surprising. However, setting a value that is too small slows down convergence considerably. Hence we will set $\alpha = \Delta_{max}/1000$ and $\beta = \alpha/10$.

In Figure 5 we see that the algorithm with the recommended parameters produces a stable result irrespective of network size or message drop rate distribution. There is a notable exception: the uniform distribution, where nodes with very small drop rates get slightly overrepresented. To see why this happens, recall that in the algorithm we made the simplifying assumption that the average update frequency is $1/\Delta_{max}$. This assumption is violated in the uniform model, where the average drop rate is very high (0.25) which noticeably increases the average update frequency.

However, in practice, due to the skewed distribution, the average drop rate is small. Second, in special environments where the average rate is high, one can approximate the average rate using suitable techniques (for example, see [3]).

Finally, in Figure 6 we show the dynamics of Δ and n with the recommended parameter settings. We observe that convergence requires approximately 500 cycles after which the periods of the nodes fluctuate in a bounded interval. The Figure also shows n as a function of time. Here we see that the variance is large. However, most importantly, it is not larger than that of the original Newscast protocol where this level of variance is normal [12].

5 Unbalanced Application Load

5.1 Algorithm

If the application at a node requests many random samples, then the node should communicate faster to refresh its view more often. Nevertheless we should mention that it is *not* a good solution to simply speed up Algorithm 1 locally. This is because in this case a fast node would inject itself into the network more often, quickly getting a disproportionate representation in the network. To counter this, we need to keep the frequency of update messages unchanged and we need to introduce extra shuffle messages without injecting new information.

To further increase the diversity of the peers to be selected, we apply a tabu list as well. Algorithm 2 implements these ideas (we show only the differences from Algorithm 1).

Shuffle messages are induced by the application when it calls the API of the peer sampling service; that is, procedure `getRandomPeer`: the node sends a shuffle message after every S random peer requests. In a practical implementation one might want to set a minimal waiting time between sending two shuffle messages. In this case, if the application requests random peers too often, then it will experience a lower quality of service (that is, a lower degree of randomness) if we decide to simply not send the shuffle message; or a delayed service if we decide to delay the shuffle message.

We should add that the idea of shuffling is not new, the Cyclon protocol for peer sampling is based entirely on shuffling [10]. However, Cyclon itself shares the same problem concerning non-uniform application load; here the emphasis is on adaptively applying *extra* shuffle messages where the sender does not advertise itself.

The tabu list is a FIFO list of fixed maximal size. Procedure `findFreshPeer` first attempts to pick a node address from the view that is not in the tabu list. If each node in the view is in the tabu list, a random member of the view is returned.

Note that the counter is reset when an incoming shuffle message arrives. This is done so as to avoid sending shuffle requests if the view has been refreshed during the waiting period of S sample requests.

Finally, procedure `shuffle` takes the two views and for each position it randomly decides whether to exchange the elements in that position; that is, no elements are removed and no copies are created.

Algorithm 2. Extending Newscast with on-demand shuffling

1: procedure <code>getRandomPeer</code>	11: procedure <code>onShuffleUpdateResponse(m)</code>
2: $p \leftarrow \text{findFreshPeer}()$	12: $\text{buffer} \leftarrow m.\text{buffer}$
3: $\text{tabuList.add}(p)$	13: $\text{counter} \leftarrow 0$
4: $\text{counter} \leftarrow \text{counter} + 1$	14: end procedure
5: if $\text{counter} = S$ then	15:
6: $\text{counter} \leftarrow 0$	16: procedure <code>onShuffleUpdate(m)</code>
7: send <code>shuffleUpdate(view)</code> to p	17: $(\text{buffer}_1, \text{buffer}_2) \leftarrow \text{shuffle}(\text{view}, m.\text{buffer})$
8: end if	18: send <code>shuffleUpdateResp(buffer₁)</code> to $m.\text{sender}$
9: return p	19: $\text{buffer} \leftarrow \text{buffer}_2$
10: end procedure	20: end procedure

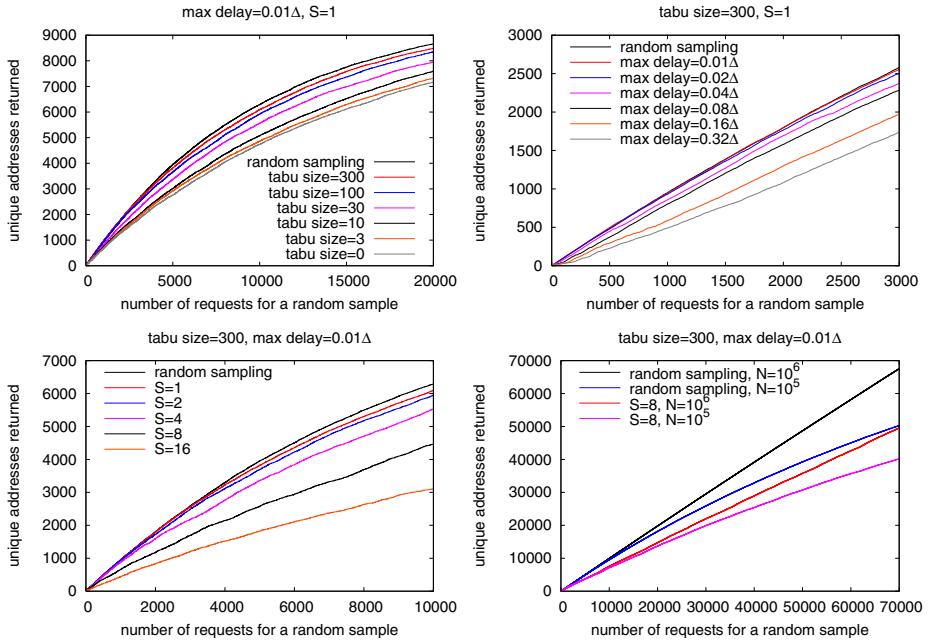


Fig. 7. Experimental results with adaptive shuffling for $N = 10^4$ if not otherwise indicated. In B/W print, lines are shown in the same line-type: the order of keys matches the order of curves from top to bottom.

5.2 Evaluation

Like in Section 4.3, we ran event-driven simulations over PeerSim. Messages were delayed using a uniform random delay between 0 and a given maximal delay.

In all the experiments, we worked with a scenario where peer sampling requests are maximally unbalanced: we assumed that the application requests samples at a high rate on one node, and no samples are requested on the other nodes. This is our worst case scenario, because there is only one node that is actively initiating shuffle requests. The other nodes are passive and therefore can be expected to refresh their own view less often.

The experimental results are shown in Figure 7. The parameters we examine are S , the size of the tabu list, network size (N) and the maximal delay. The values of the maximal delay go up to 0.3Δ , which is already an unrealistically long delay if we consider that practical values of Δ are high, around 10 seconds or more. In fact, in the present adaptive version Δ can be much higher since randomness is provided by the shuffling mechanism.

First of all, we notice that for many parameter settings we can get a sample diversity that is almost indistinguishable from true random sampling, especially when S and the maximal delay are relatively small. For $S = 2$ and $S = 4$ the returned samples are still fairly diverse, which permits one to reduce the number of extra messages by a factor of 2 or even 4. The tabu list is “free” in terms of network load, so we can set high values, although beyond a certain point having higher values appears to make no difference.

The algorithm is somewhat sensitive to extreme delay, especially during the initial sample requests. This effect is due to the increased *variance* of message arrival times, since the number of messages is unchanged. Due to variance, there may be large intervals when no shuffle responses arrive. This effect could be alleviated via queuing the incoming shuffle responses and applying them in equal intervals or when the application requests a sample.

Since large networks are very expensive to simulate, we will use just one parameter setting for $N = 10^5$ and $N = 10^6$. In this case we observe that for large networks randomness is in fact slightly better, so the method scales well.

6 Conclusions

In this paper we identified two cases where the non-uniformity of the environment can result in a serious performance degradation of gossip-based peer sampling protocols, namely non-uniform message drop rates and an unbalanced application load.

Concentrating on Newscast we offered solutions to these problems based on a statistical approach, as opposed to relatively heavy-weight reliable measurements, reliable transport, or failure detectors. Nodes simply observe the local events and based on that they modify the local parameters. As a result, the system converges to a state that can handle non-uniformity.

The solutions are cheap: in the case of symmetric message drop rates we require no extra control messages at all. In the case of application load, only the local node has to initiate extra exchanges proportional to the application request rate; but for any sampling protocol that maintains only a constant-size state this is a minimal requirement.

References

1. Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., Terry, D.: Epidemic algorithms for replicated database maintenance. In: Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC 1987), Vancouver, British Columbia, Canada, pp. 1–12. ACM Press, New York (1987)
2. Kempe, D., Dobra, A., Gehrke, J.: Gossip-based computation of aggregate information. In: Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2003), pp. 482–491. IEEE Computer Society Press, Los Alamitos (2003)
3. Jelassi, M., Montresor, A., Babaoglu, O.: Gossip-based aggregation in large dynamic networks. ACM Transactions on Computer Systems 23(3), 219–252 (2005)
4. Bonnet, F., Kermarrec, A.-M., Raynal, M.: Small-world networks: From theoretical bounds to practical systems. In: Tovar, E., Tsigas, P., Fouchal, H. (eds.) OPODIS 2007. LNCS, vol. 4878, pp. 372–385. Springer, Heidelberg (2007)
5. Patel, J.A., Gupta, I., Contractor, N.: JetStream: Achieving predictable gossip dissemination by leveraging social network principles. In: Proceedings of the Fifth IEEE International Symposium on Network Computing and Applications (NCA 2006), Cambridge, MA, USA, July 2006, pp. 32–39 (2006)
6. Voulgaris, S., van Steen, M.: Epidemic-style management of semantic overlays for content-based searching. In: Cunha, J.C., Medeiros, P.D. (eds.) Euro-Par 2005. LNCS, vol. 3648, pp. 1143–1152. Springer, Heidelberg (2005)

7. Jelasity, M., Babaoglu, O.: T-man: Gossip-based overlay topology management. In: Brueckner, S.A., Di Marzo Serugendo, G., Hales, D., Zambonelli, F. (eds.) ESOA 2005. LNCS, vol. 3910, pp. 1–15. Springer, Heidelberg (2006)
8. Zhong, M., Shen, K., Seiferas, J.: Non-uniform random membership management in peer-to-peer networks. In: Proc. of the IEEE INFOCOM, Miami, FL (2005)
9. Allavena, A., Demers, A., Hopcroft, J.E.: Correctness of a gossip based membership protocol. In: Proceedings of the 24th annual ACM symposium on principles of distributed computing (PODC 2005), Las Vegas, Nevada, USA. ACM Press, New York (2005)
10. Voulgaris, S., Gavridia, D., van Steen, M.: CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays. *Journal of Network and Systems Management* 13(2), 197–217 (2005)
11. Eugster, P.T., Guerraoui, R., Handurukande, S.B., Kermarrec, A.M., Kouznetsov, P.: Lightweight probabilistic broadcast. *ACM Transactions on Computer Systems* 21(4), 341–374 (2003)
12. Jelasity, M., Voulgaris, S., Guerraoui, R., Kermarrec, A.M., van Steen, M.: Gossip-based peer sampling. *ACM Transactions on Computer Systems* 25(3), 8 (2007)
13. Yalagandula, P., Dahlin, M.: Shruti: A self-tuning hierarchical aggregation system. In: IEEE Conference on Self-Adaptive and Self-Organizing Systems (SASO), pp. 141–150. IEEE Computer Society, Los Alamitos (2007)
14. Zhang, Y., Duffield, N.: On the constancy of Internet path properties. In: Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement (IMW 2001), pp. 197–211. ACM Press, New York (2001)
15. PeerSim, <http://peersim.sourceforge.net/>

Exploring the Feasibility of Reputation Models for Improving P2P Routing under Churn*

Marc Sàncchez-Artigas, Pedro García-López, and Blas Herrera

Universitat Rovira i Virgili, Catalonia, Spain

{marc.sanchez,pedro.garcia,blas.herrera}@urv.cat

Abstract. Reputation mechanisms help peer-to-peer (P2P) networks to detect and avoid unreliable or uncooperative peers. Recently, it has been discussed that routing protocols can be improved by conditioning routing decisions to the past behavior of forwarding peers. However, churn — the continuous process of node arrival and departure — may severely hinder the applicability of rating mechanisms. In particular, short lifetimes mean that reputations are often generated from a small number of transactions.

To examine how high rates of churn affect rating mechanisms, this paper introduces an analytical model to compute at which rate transactions has to be performed so that the generated reputations are sufficiently reliable. We then propose a new routing protocol for structured P2P systems that exploits reputation to improve the decision about which neighbor choose as next hop. Our results demonstrate that routing algorithms can extract substantial benefits from reputations even when peer lifetimes are short.

1 Introduction

Resilience of structured Peer-to-Peer (P2P) networks under churn has attracted significant attention during the last few years. One of the fundamental problems of these systems is the ability to locate resources as peers join and fail at a high rate of churn. Churn may cause the staleness of immediate neighbors, and hence importantly deteriorate the performance of the underlying routing protocol. This has led to the development of reputation models such as the Feedback Forwarding Protocol (FFP) [1] and the model of Artigas et. al. [2] to detect reliable message forwarders. In these models, each peer captures evidence to quantify the behavior of its immediate neighbors. Based on this evidence, which corresponds to either success or failure of message delivery, each forwarder can evaluate whether it did a satisfactory routing job or not, and select as next hop the neighbor more likely to deliver the message.

Unfortunately, yet, existing models suffer from the same syndrome they want to ameliorate. Since reputations assess trustworthiness using historical feedback, short peer lifetimes mean that reputation values may not be sufficiently accurate,

* This research was partially funded through project P2PGRID, TIN2007-68050-C03-03, of the Ministry of Education and Science, Spain.

which will result in poor routing decisions and inconstant performance. One way to address this problem was proposed by Swamynathan et. al. [3]. Through the use of proactive reputations, they developed a mechanism for generating accurate reputations for newcomers and those peers with short lifetimes. Their idea was to allow peers to proactively initiate transactions for the mere purpose of generating reliable ratings, and thus improve the overall quality of routing decisions.

While very promising, the proactive computation of ratings requires peers to determine under which rate of churn reputations should be proactively generated. Otherwise, the additional traffic injected into the network might drain away the benefits of a better routing service.

To examine this question, we make the following key contributions:

- We introduce an analytical model to derive at which rate feedback has to be collected in order to generate accurate reputations. Specifically, we consider the distribution of *residual neighbor lifetime*, which plays a fundamental role in the feasibility analysis throughout this work. With our model, we observe that depending on the tail-weight of the lifetime distribution, the probability of generating a reliable rating can be made arbitrarily large without incurring excessive overhead.
- We propose a new routing protocol for structured P2P systems that exploits reputation to improve the decision about which neighbor choose as next hop. We use a Bayesian reputation model, specifically a beta model, for reputation evolution. The main characteristic of our protocol is that progression through the identifier space is not completely greedy; instead, it selects next hop peers that are presumably stable, though the average path length increases slightly.

The remaining of the paper proceeds as follows. In Section 2, we introduce the model for studying the effects of churn. In Section 3, we provide some analytical results on the feasibility of reputations systems under churn. Section 4 describes our novel optimization technique. Simulations results are presented in Section 5. Finally, Section 6 discusses related work followed by the conclusions in Section 7.

2 System Model

In P2P networks, each peer creates links to other d peers when joining the system, where d may be a constant or a function of system size such as Chord [4]. Further, each peer detects and repairs link failures in order to stay connected. Under some general conditions [5], link dynamics can be represented as an ON/OFF process. Specifically, a link is ON whenever the adjacent neighbor to the link is alive, and OFF, when it has departed and its failure is in the process of being detected and repaired.

In some P2P networks, specially unstructured systems, links do no switch to other peers during ON time (i.e., remain connected to the same neighbors until they fail). Then, link durations are simply residual lifetimes of original neighbors. In structured P2P networks, however, this assumption does not usually hold. For example, deterministic DHTs (e.g. Chord) actively switch links to new neighbors

before the adjacent ones leave the system. For instance, in Chord, a switch occurs when a joining peer takes ownership of a link by becoming the new successor of the corresponding neighbor pointer.

In this paper, we assume that links do not switch before the current neighbors fail. The reason is quite subtle. In general, a link switch requires recomputing the reputation associated with that link, since very often there is no prior knowledge about the new neighbor. In practice this increases the odds of short-term ratings, for which there is no compelling reason to allow peers to switch between live links other than preserving the original routing properties, on the average path length principally. In DHTs such as Pastry, Tapestry or Kademlia, this assumption has no negative repercussions. In Chord, the adoption of this assumption randomizes routing, for we see no sufficient reason to actively switch links to new neighbors.

For the churn model, we adopt the model of Leonard et. al. [5]. More precisely, we consider that each joining peer is assigned a random lifetime drawn from some distribution $F(t)$, which reflects the amount of time the peer stays in the system. Since it is almost impossible for a newcomer to pick its neighbors such that their arrival times are exactly the same as herself, neighbor selection is performed over the peers already present in the network. These nodes have been online for some random amount of time, which means that residual lifetime has to be considered.

We now formalize the notion of residual lifetime. Let R_i denote the remaining lifetime of neighbor i when the newcomer joined the system. Assuming that the system has reached stationarity (i.e., peers joining the system at different times of the day or month have their lifetimes drawn from the same distribution $F(t)$), the cumulative density function (CDF) of residual lifetime is given by [6]:

$$F_R(t) = \Pr(R_i < t) = \frac{1}{\mathbb{E}[L_i]} \int_0^t (1 - F(z)) dz \quad (1)$$

where $F(t)$ is the CDF of neighbor's i lifetime L_i , and $\mathbb{E}[L_i]$ its expectation. As in [5], we will experiment with two lifetime distributions:

- *Exponential lifetime*, which has been observed in PlanetLab, with the CDF given by $F(t) = 1 - e^{-\lambda t}$, $\lambda > 0$; and
- *Pareto lifetime*, which has been reported by several works (see [7] for details) to provide a tight fit to the lifetime distributions found in real P2P systems. Its CDF is $F(t) = 1 - (1 + \frac{t}{\beta})^{-\alpha}$, $\alpha > 1$, where α represents the heavy-tailed degree and β is a scaling parameter.

For exponential lifetimes, the residual lifetimes are also exponential thanks to the memoryless property of $F(t)$, i.e., $F_R(t) = 1 - e^{-\lambda t}$. However, for Pareto lifetimes, the residuals are *more heavy-tailed* and exhibit shape parameter $\alpha - 1$, i.e., $F_R(t) = 1 - (1 + \frac{t}{\beta})^{1-\alpha}$, $\alpha > 1$.

Another assumption is as follows. While online, we consider that the number of transactions issued for each neighbor i follows a homogeneous Poisson process with rate μ_i . This implies that the probability that exactly x transactions occur in time interval $(0, t]$ can be written as

$$\Pr(X_i(t) = x | \mu_i) = \frac{(\mu_i t)^x e^{-\mu_i t}}{x!}, \quad x = 0, 1, 2, \dots \quad (2)$$

Since the sum of independent Poisson processes is again a Poisson process, then total number of transactions, $X = \sum_{i=1}^d X_i$, is a Poisson process with rate equal to $\mu = \sum_{i=1}^d \mu_i$. Finally, we should mention that the above assumption does not hold when the transaction rate changes over time. This is especially visible when reputations are applied to routing. Since the computation of reputations is made based on the success or failure of message delivery, reliable neighbors will receive more messages to transmit. This implies that the transaction rate of all neighbors will vary over time. More precisely, the transaction rate of neighbor i will be equal to $\Lambda_i(t) = \int_0^t \mu_i(z) dz$, making the Poisson process non-homogeneous. However, since all non-homogeneous Poisson processes can be reduced to the homogeneous case, we only considered homogeneity to keep the derivations tractable.

3 Feasibility Analysis

In this section, we focus on the probability that a reliable reputation can be built before the corresponding neighbor leaves the system. We call this probability the *probability of feasibility* π . In our analysis, a reputation is said to be reliable when the level of confidence in its value is sufficiently high. In practice, this means that the minimum number of interactions, say m , over which it makes sense to assume stationary behavior has occurred.

More formally, consider the i^{th} neighbor of an arbitrary node in the network. Its reputation is reliable only if the number of messages $X_i(t_{OFF})$ routed through i is greater than m before it decides to abandon the system at time t_{OFF} . Then,

$$\pi_i = \int_0^\infty \Pr(X_i(z) \geq m) f_R(z) dz \quad (3)$$

where $f_R(z)$ is the the probability density function (PDF) of the residual lifetime.

Theorem 1. *For exponential lifetimes,*

$$\pi_i = \left(\frac{\mu_i}{\mu_i + \lambda} \right)^m \quad (4)$$

and for Pareto lifetimes with $\alpha > 1$:

$$\pi_i = 1 + e^{\mu_i \beta} (1 - \alpha) \sum_{\omega=0}^{m-1} \frac{\mu_i^\omega}{\omega!} \beta^\omega G(\omega, \mu_i, \alpha, \beta) \quad (5)$$

$$\text{where } G(\omega, \mu_i, \alpha, \beta) = \sum_{n=0}^{\omega} (-1)^{\omega-n} \binom{\omega}{n} (\mu_i \beta)^{\alpha-n-1} \Gamma(1 - \alpha + n, \mu_i \beta).$$

Proof. For exponential lifetimes, we have:

$$\pi_i = \int_0^\infty \left(\sum_{\omega=m}^{\infty} \frac{(\mu_i x)^\omega e^{-\mu_i x}}{\omega!} \right) \lambda e^{-\lambda x} dx$$

$$\begin{aligned}
&= \lambda \sum_{\omega=m}^{\infty} (\mu_i)^\omega \int_0^\infty \frac{x^\omega e^{-(\mu_i+\lambda)x}}{\omega!} dx \\
&= \lambda \sum_{\omega=m}^{\infty} \frac{(\mu_i)^\omega}{(\mu_i+\lambda)^{\omega+1}} \int_0^\infty \frac{(\mu_i+\lambda)^{\omega+1} x^\omega e^{-(\mu_i+\lambda)x}}{\omega!} dx
\end{aligned} \tag{6}$$

Now noting that the integrand in (6) is an Erlang- $(\omega+1)$ PDF,

$$\pi_i = \lambda \sum_{\omega=m}^{\infty} \frac{(\mu_i)^\omega}{(\mu_i+\lambda)^{\omega+1}} = \frac{\lambda}{(\mu_i+\lambda)} \sum_{\omega=m}^{\infty} \left(\frac{\mu_i}{\mu_i+\lambda} \right)^\omega = \left(\frac{\mu_i}{\mu_i+\lambda} \right)^m \tag{7}$$

For Pareto lifetimes, we have:

$$\begin{aligned}
\pi_i &= \int_0^\infty \left(1 - \sum_{\omega=0}^{m-1} \frac{(\mu_i x)^\omega e^{-\mu_i x}}{\omega!} \right) \frac{-(1-\alpha)}{\beta} \left(1 + \frac{x}{\beta} \right)^{-\alpha} dx = \\
&= -(1-\alpha) \beta^{\alpha-1} \int_0^\infty (\beta + x)^{-\alpha} dx + \\
&+ (1-\alpha) \beta^{\alpha-1} \sum_{\omega=0}^{m-1} \frac{\mu_i^\omega}{\omega!} \int_0^\infty x^\omega e^{-\mu_i x} (\beta + x)^{-\alpha} dx
\end{aligned}$$

We note that $\alpha > 1$, $\beta > 0$ and $-(1-\alpha) \beta^{\alpha-1} \int_0^\infty (\beta + x)^{-\alpha} dx = 1$, then

$$\begin{aligned}
\pi_i &= 1 + (1-\alpha) \beta^{\alpha-1} \sum_{\omega=0}^{m-1} \frac{\mu_i^\omega}{\omega!} \int_0^\infty x^\omega e^{-\mu_i x} (\beta + x)^{-\alpha} dx = \\
&= 1 + (1-\alpha) \beta^{\alpha-1} \sum_{\omega=0}^{m-1} \frac{\mu_i^\omega}{\omega!} \frac{1}{\beta^\alpha} \int_0^\infty \frac{x^\omega}{\left(1 + \frac{x}{\beta}\right)^\alpha} e^{-\mu_i x} dx
\end{aligned}$$

By the change of variable $1 + \frac{x}{\beta} = t$, we have

$$\pi_i = 1 + (1-\alpha) \beta^{\alpha-1} \sum_{\omega=0}^{m-1} \frac{\mu_i^\omega}{\omega!} e^{\mu_i \beta} \beta^{\omega-\alpha+1} \int_1^\infty \frac{(t-1)^\omega}{t^\alpha} e^{-\mu_i \beta t} dt$$

We simplify and we use the binomial identity of Newton

$$\begin{aligned}
\pi_i &= 1 + e^{\mu_i \beta} (1-\alpha) \sum_{\omega=0}^{m-1} \frac{\mu_i^\omega}{\omega!} \beta^\omega \int_1^\infty \frac{\sum_{n=0}^{\omega} \binom{\omega}{n} t^n (-1)^{\omega-n}}{t^\alpha} e^{-\mu_i \beta t} dt = \\
&= 1 + e^{\mu_i \beta} (1-\alpha) \sum_{\omega=0}^{m-1} \frac{\mu_i^\omega}{\omega!} \beta^\omega \int_1^\infty e^{-\mu_i \beta t} \left(\sum_{n=0}^{\omega} \binom{\omega}{n} t^{n-\alpha} (-1)^{\omega-n} \right) dt =
\end{aligned}$$

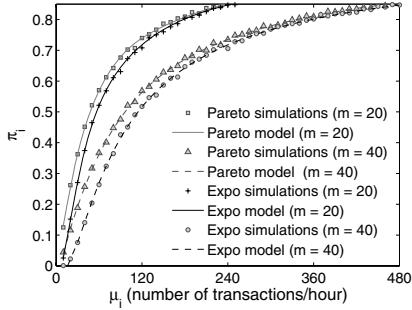


Fig. 1. Comparison of probability π_i from Theorem 1 to simulation results as rate μ_i varies

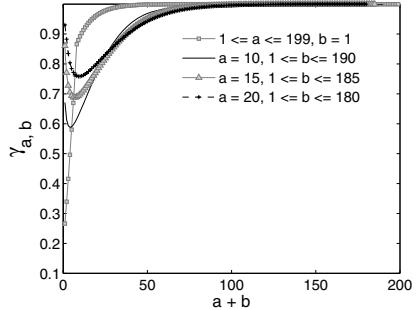


Fig. 2. Confidence (see eq. (8)) versus the number of observations.

The finite series commutes with the integral

$$\begin{aligned} \pi_i &= 1 + e^{\mu_i \beta} (1 - \alpha) \sum_{\omega=0}^{m-1} \frac{\mu_i^\omega}{\omega!} \beta^\omega \sum_{n=0}^{\omega} (-1)^{\omega-n} \binom{\omega}{n} \int_1^\infty t^{n-\alpha} e^{-\mu_i \beta t} dt = \\ &= 1 + e^{\mu_i \beta} (1 - \alpha) \sum_{\omega=0}^{m-1} \frac{\mu_i^\omega}{\omega!} \beta^\omega \sum_{n=0}^{\omega} (-1)^{\omega-n} \binom{\omega}{n} \int_1^\infty \frac{e^{-\mu_i \beta t}}{t^{\alpha-n}} dt \end{aligned}$$

The generalized exponential integral function, E_n , is related with the incomplete gamma function, with $x > 0$, in the following way: $\int_1^\infty \frac{e^{-xt}}{t^a} dt = x^{a-1} \Gamma(1-a, x)$. Since $\mu_i \beta > 0$, we finally have $\pi_i = 1 + e^{\mu_i \beta} (1 - \alpha) \sum_{\omega=0}^{m-1} \frac{\mu_i^\omega}{\omega!} \beta^\omega G(\omega, \mu_i, \alpha, \beta)$. \square

From Theorem 1, it is easy to obtain the values of μ_i which make π_i to be within a small window $[\varepsilon, 1]$, where ε is a small fraction of error. More exactly, by solving inequality $\pi_i(\mu_i, m) = \int_0^\infty \left(\sum_{\omega=m}^\infty \frac{(\mu_i z)^\omega e^{-\mu_i z}}{\omega!} \right) f_R(z) dz \geq \varepsilon$ for variable μ_i , one can easily write the feasibility condition for the reputation system, and determine if the normal execution of the application is sufficient to generate reliable ratings. One immediate corollary of Theorem 1 is thus:

Corollary 1. *For exponential lifetimes, the feasibility condition is $\mu_i \geq \frac{1}{(\varepsilon^{\frac{1}{m}} - 1)} \lambda$, where $1 - \varepsilon$ is the tolerance level for the fraction of reliable ratings in the system.*

For the Pareto part of Theorem 1, there is no closed-form solution for π_i . To tackle this problem, inequality $\pi_i = 1 + e^{\mu_i \beta} (1 - \alpha) \sum_{\omega=0}^{m-1} \frac{\mu_i^\omega}{\omega!} \beta^\omega G(\omega, \mu_i, \alpha, \beta) \geq \varepsilon$ can be numerically solved using the Regula Falsi method¹, with $G(\omega, \mu_i, \alpha, \beta)$ defined as Theorem 1.

The accuracy of eqs. (4)-(5) is illustrated in Figure 1, which plots π_i for four simulations together with those predicted by us. As done in [5], parameters α and

¹ Faster numerical methods such as Newton-Raphson exhibit convergence failures.

λ has been selected so that $E[L_i]$ is 0.5 hours for both lifetime distributions. The scaling parameter β has been set to 1 in the Pareto distribution. For simulations, we used a hypothetical system with $N = 1,000$ peers and degree $d = 10$, in which each failed peer was immediately replaced by a fresh peer. As shown in the figure, simulations agree with predicted results very well.

4 The Reliable Forwarder (\mathcal{RF}) Scheme

In this section, we examine a mechanism to optimize routing paths. Our goal is to optimize routing paths so that peers collectively select the routes that are likely to successfully deliver messages. This mechanism, named the *Reliable Forwarder* (\mathcal{RF}) scheme, proposes a reputation system based on the Beta reputation model to tolerate churn in a fully distributed manner. It is important to note here that the objective of this research is not to propose the optimal routing protocol but rather to provide an experimental study on what can be achieved in DHTs, from the reputation viewpoint.

Overview. Although the \mathcal{RF} scheme is applicable to almost all structured P2P networks, we employ Chord [4] for illustration purposes. In this work, we assume that the reader is familiar with Chord. Following convention, we will indistinctly refer to a peer and the identifier of a peer by p . The intuition of the protocol is as follows: in the original proposal, Chord greedily routes towards the destination in decreasing distances. However, this order is in no way essential to the correctness of routing. This provides some room for selecting stable peers on delivery paths. Specifically, the idea is to loosen the rule of strictly selecting the neighbor closest to the destination in the following way:

Assume that the remaining distance from p to the destination is d_p . Let $j(p)$ be the index of p 's neighbor that most closely precedes the destination, i.e., $j(p)$ is the largest integer verifying that $2^{j(p)} + h \leq d_p$. Under our scheme, p considers the $j(p)$ neighbors in increasing order of distance w.r.t. the destination until one neighbor is picked randomly. The election is done in proportion to its reputation, as *its reputation value captures how well this neighbor forwards messages*.

As expected, reputations are built by injecting new messages into the system, and tracking continuously the feedback received for each message. Obviously, this process re-adjusts the paths and ameliorates the effects of churn.

Feedback. To collect evidence to asseverate either success or failure of message delivery, we make use of the Forward Feedback Protocol (FFP) introduced in [1]. In this protocol, each routed message is followed on its routing path by a feedback message. A transaction is always initiated by the source by sending out a message for some (random) key, which in addition starts a timeout. Thus, after a timeout or a response, the source obtains a feedback that can use to notify each peer along the routing path about the success or failure of message delivery. More precisely, the first feedback message is sent immediately by the source upon reception of a response. Then on, the feedback is forwarded following exactly the same routing path the original message followed. With this feedback, each intermediate

router along the path can learn whether it did a good job or not, and feed the reputation system with valuable information. The use of FFP was motivated by two reasons:

- *Fast convergence*: since observations are shared with the peers along routing paths, the system can converge to a state in which there are few lost messages faster than when a peer produces the reputations for its neighbors exclusively on its own. Of course, we assume that peers are honest and cooperative.
- *Low overhead*: the benefits of such a fast convergence comes only at the price of doubling the routing cost; thus the overhead is $\mathcal{O}(\log n)$ messages.

Representation and Calculation (Beta Reputation Model [8]). Concretely, each peer characterizes the behavior of its neighbor i as follows. It considers that there is a parameter θ such that neighbor i is capable of finding a reliable routing path with probability θ . The parameter θ is unknown, and each peer models this uncertainty by assuming that θ itself is drawn according to a distribution, called the *prior distribution*, which is updated with each new transaction. In Bayesian analysis, the distribution $\text{Beta}(a, b)$ is commonly selected as a prior distribution for random variables that take on real values in the interval $[0, 1]$. For this reason we use distribution $\text{Beta}(a, b)$ in our model.

The standard Bayesian procedure is as follows. Initially, the prior is $\text{Beta}(1, 1)$, i.e., the uniform distribution on $[0, 1]$; this represents absence of knowledge about the possible values for θ . Then, when $s + f$ transactions are performed, say with s successful deliveries and f failures, the prior is updated according to expressions $a = a + s$ and $b = b + f$. If θ , the true unknown parameter, is constant, then after a sufficient number of transactions, $a \approx \theta n$, $b \approx (1 - \theta)n$, and $\text{Beta}(a, b)$ becomes close to a Dirac at θ . The advantage of using the Beta distribution is that it only requires two parameters that are continuously updated as new transactions are made. Under this model, we compute reputations as follows:

Definition 1. Consider a peer p with d neighbors. Under the standard Bayesian Beta model, we define the reputation $T_{a,b}^i$ of neighbor i at a time t as the expected value of the $\text{Beta}(a, b)$, where $a + b$ is the total number of transactions made with neighbor i up to time t . Formally, $T_{a,b}^i = \mathbb{E}[\text{Beta}(a, b)] = \frac{a}{a+b}$.

Notice that our definition of reputation not only includes first-hand information, i.e., the feedback corresponding to the transactions started individually by p , but also the opinions of other online peers in the system. More precisely, their opinion is captured by increasing either parameter a or b depending on the feedback sent by the source peer through the FFP (see preceding section for details).

On its own, $T_{a,b}^i$ does not distinguish between the cases where a link is reliable from the cases where it is not. This uncertainty raises the necessity to accurately gauge the confidence in $T_{a,b}^i$ to avoid making hasty decisions, for which we define a confidence metric, $\gamma_{a,b}$, as the posterior probability that the current value of θ lies within an acceptable level of error ϵ about $T_{a,b}^i$. For instance, if the number of transactions is constant, a larger value of ϵ causes a node to be more confident

in its $T_{a,b}^i$ than a lower value of ϵ . The first part of eq. (8) comes from the work of Teacy et. al. in [9].

$$\gamma_{a,b}^i = \frac{\int_{T_{a,b}^i - \epsilon}^{T_{a,b}^i + \epsilon} \theta^{a-1} (1-\theta)^{b-1} d\theta}{\int_0^1 t^{a-1} (1-t)^{b-1} dt} = I_{T_{a,b}^i + \epsilon}(a, b) - I_{T_{a,b}^i - \epsilon}(a, b) \quad (8)$$

where $I_x(\alpha, \beta)$ is the regularized incomplete beta function. Figure 2 illustrates how confidence grows when the number of transactions increases, for different values of the parameters a and b of the Beta distribution.

In the last section, we examined how the transaction rate affects the reliability of ratings. For the case of the Beta reputation model, $\gamma_{a,b}^i$ can be used to calculate the threshold on the number of transactions. However, since an exact expression of eq. (8) is hard to develop in closed-form (it depends on $T_{a,b}^i$), we have opted here to set a threshold γ on $\gamma_{a,b}^i$ directly. Specifically, γ represents the minimum level of confidence to consider $T_{a,b}^i$ as a reliable reputation. Figure 3 illustrates the relationship between parameters m and θ for 4 combinations of γ and ϵ . As seen in the figure, a small increase in ϵ results in a sharp increase in m .

4.1 Routing-Decision Process

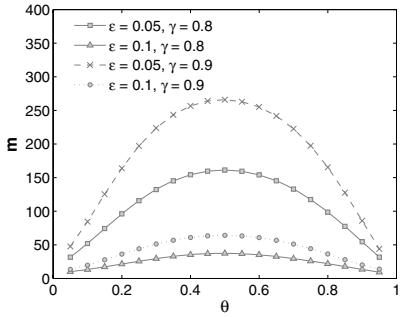
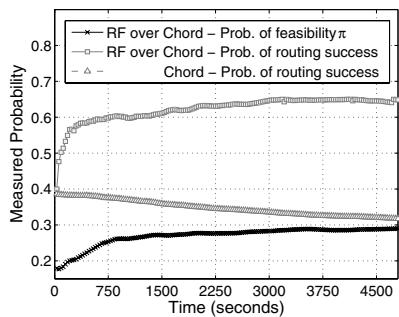
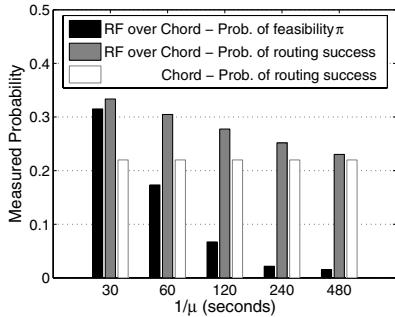
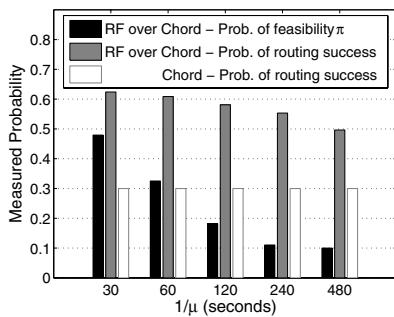
With the confidence metric, the decision-making process works as follows. First, the posterior is calculated based on all the available knowledge. For each neighbor i , this is achieved by updating $T_{a,b}^i$ and $\gamma_{a,b}^i$ as explained above. Then, each router p takes a decision in the following way:

1. Peer p considers up to $j(p)$ alternatives to forward the message, where $j(p)$ is the index of the neighbor that most closely precedes the destination.
2. If neighbor $j(p)$ is the destination, then the decision-making process stops.
3. Otherwise, p begins with neighbor $j(p)$ and performs a Bernoulli trial, $X_{j(p)}$, that equals 1 ($X_{j(p)} = 1$ means a success) with probability

$$w_{j(p)}^p = \Pr(X_{j(p)} = 1) = T_{a,b}^{j(p)} \begin{cases} \frac{\gamma_{a,b}^{j(p)}}{\gamma} & \text{if } \gamma_{a,b}^{j(p)} < \gamma \\ 1 & \text{otherwise} \end{cases} \quad (9)$$

where γ is the minimum level of confidence to deem $T_{a,b}^{j(p)}$ reliable. Intuition of this formula is as follows. In the preceding section we derived an expression to compute the confidence on a reputation. Because an optimal routing decision is not the election of a neighbor only in proportion to its reputation but also on how reliable the reputation is, (9) includes a linear drop-off in reliability, so that neighbors with reliable ratings are more likely to be selected.

4. If trial $X_{j(p)} = 0$ (fails), then steps 2–3 are repeated for neighbor $i = j(p) - 1$. If this trial fails, steps 2–3 are again repeated but for neighbor $i = j(p) - 2$. This is repeated until some trial is successful.
5. If eventually $i = 0$, then the jump is always performed. Note that this jump corresponds to the immediate successor of p on the identifier space.

**Fig. 3.** Relationship between m and θ **Fig. 4.** Performance of the \mathcal{RF} scheme as a function of time ($1/\mu = 30$ seconds)**Fig. 5.** Performance of the \mathcal{RF} scheme for exponential lifetimes ($\lambda = 2$)**Fig. 6.** Performance of the \mathcal{RF} scheme for Pareto lifetimes ($\alpha = 2$; $\beta = 1$)

Notice that probabilities w_i^p are the only state each peer has to maintain for churn-aware routing. It grows $\mathcal{O}(\log n)$ per peer, scaling well with the number of peers in the network.

Example 1. We focus on the hypothetical case that $j(p) = 2$. As explained above, this means that the \mathcal{RF} scheme starts with neighbor $j(p) = 2$, which corresponds to a jump of length at least 4 on the Chord circle. This neighbor is selected with probability w_2^p . If not possible, then p tries to choose neighbor 1 with probability $(1 - w_2^p)w_1^p$. Finally, it will consider neighbor 0 (successor of p) if neither neighbor 2 nor neighbor 1 are chosen. This neighbor is always chosen to guarantee progress towards the destination.

5 Simulated Performance Results

In this section, we concisely describe the simulation environment followed by the performance evaluation of our scheme in various scenarios.

5.1 Simulator and Environment

We designed an event-driven simulator with the Chord and our scheme². Initially, we constructed a Chord network consisting of $n = 1,000$ peers³. Each peer lived for a random duration L_i and then left ungracefully (without notifying others). In consequence, delivery failures were caused by peer departures. To repair the failures, we considered that all peers periodically invoked functions `stabilize()` and `fix_fingers()`. Each call to `fix_fingers()` updated a single neighbor. The period T_{fix} between two successive calls to `fix_fingers()` was set to $T_{\text{fix}} = 15$ seconds.

To prevent the network from depleting to zero, we assumed that when a live peer departed it was immediately replaced by a fresh peer with random lifetime L_i ; the exact arrival process was not essential and had no impact on the results. The two lifetime distributions discussed in this paper were tested. Specifically, parameters α and λ were selected so that $E[L_i] = 0.5$ hours for both distributions and the scaling parameter β was set to 1 in the Pareto $F(t)$.

In addition, we assumed that all live peers injected messages into the network according to a Poisson process with rate μ . Since the rate of successful message delivery is our focus, the simulator did not model real features such as network delay. As a result, we assumed the latency incurred by each hop to be uniformly distributed between [100, 200] milliseconds. Regarding rating parameters, we set γ and ϵ to 0.8 and 0.1, respectively.

5.2 Experiments

To assess how the value of μ affects the performance of the \mathcal{RF} scheme, we ran several simulations varying this parameter.

Figures 5 and 6 show the probability of successfully routing a message along with the mean fraction of reliable reputations. It can be clearly inferred that our scheme improves considerably the tolerance of standard Chord against churn. In the Pareto case (Figure 5), for instance, with an average waiting time between message arrivals of $\frac{1}{\mu} = 30$ seconds, our scheme doubles the fraction of successful deliveries. Similar observations can be made in the exponential case (Figure 6). Compared with the Pareto case, the exponential case exhibits a larger number of unsuccessful message deliveries. (1) helps us to understand why. According to this expression, the residual exponential distribution remains exponential, while the Pareto case becomes more *heavy-tailed* and indeed exhibits shape parameter $\alpha - 1 = 2$. Since the fraction of long-lived peer is higher in the Pareto, our scheme can thus more easily discover reliable routing paths. Even when $1/\mu$ is high, our scheme is able to improve Chord.

Another interesting insight is that although there exists a direct relationship between π and the proportion of successful deliveries, such a relationship is not linear. This means that our scheme is able to deliver messages in the case that

² We built a lightweight version of PlanetSim (<http://www.planetsim.net/>) in Python.

³ The network size was limited to 1,000 due to high computational cost of simulations.

reputations are not sufficiently reliable. Figure 4 shows how these two parameters evolve over time. As can be seen in the figure, the slope of the π line is less abrupt at the beginning, which suggests that calculating reliable ratings is more costly than improving routing under a Bayesian framework.

6 Related Work

To mitigate the impacts of churn, a number of solutions have been proposed. In [10], Rhea et. al. proposed to handle churn by reactive or periodic failure recovery, accurate calculation of routing timeouts and proximity neighbor selection. Later, Dabek et al. [11] explored a wide range of solutions, which included iterative and recursive routing, replication and erasure coding, in their implementation of the DHash system. Another example is [12], where Krishnamurthy et al. analyzed the performance of Chord under churn by using a master-equation-based approach. Our work differs in that we consider distributed rating mechanisms as a substrate to help routing schemes to better handle churn. The unique works we are aware of that attempt to collectively optimize routing paths are those from Galuba et. al. [1] and Artigas et. al. [2], but they do not consider the way that churn affects the feasibility of their solutions; rather, they mainly focus on the problem of how to minimize message dropping by node selection.

Finally, we should mention the work of Marti et. al. [13]. Their idea was to use an external social network to improve the rate of delivered messages. Each peer sent its messages through the nodes it knew personally or were friends of friends. So, they implicitly were the first to improve the quality of paths. Compared with us, their system depends on data (social links) that may not be always available.

7 Conclusions

In this paper, we have explored how high rates of churn affect rating mechanisms. We have provided analytical expressions to appraise the feasibility of distributed ratings systems as one way to improve message delivery in peer-to-peer networks.

Furthermore, we have introduced a novel scheme to optimize route selection. The scheme chooses reliable forwarders based on a Bayesian reputation approach. It is fully distributed and no agreement is necessary. Our early simulation results report promising benefits of reputation, although it cannot be assumed in general that reputations are reliable.

References

1. Galuba, W., et al.: Authentication-free fault-tolerant peer-to-peer service provisioning. In: DBISP2P 2007. Springer, Heidelberg (2007)
2. Sànchez-Artigas, M., García-López, P., Skarmeta, A.F.G.: Secure forwarding in dHTs - is redundancy the key to robustness? In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 611–621. Springer, Heidelberg (2008)

3. Swamynathan, G., et al.: Exploring the feasibility of proactive reputations. *Concurrency and Computation: Practice and Experience* 20(2), 155–166 (2008)
4. Stoica, I., et al.: Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.* 11(1), 17–32 (2003)
5. Leonard, D., Rai, V., Loguinov, D.: On lifetime-based node failure and stochastic resilience of decentralized peer-to-peer networks. In: *ACM SIGMETRICS*, pp. 26–37 (2005)
6. Prabhu, N.: *Stochastic Processes; Basic Theory and its Applications*. Macmillan, New York (1965)
7. Sen, S., Wang, J.: Analyzing peer-to-peer traffic across large networks. *IEEE/ACM Trans. Netw.* 12(2), 155–166 (2004)
8. Jøsang, A., Ismail, R.: The beta reputation system. In: *15th Bled Electronic Commerce Conference e-Reality: Constructing the e-Economy* (2002)
9. Teacy, W., et al.: Coping with inaccurate reputation sources: experimental analysis of a probabilistic trust model. In: *AAMAS 2005*, pp. 997–1004. ACM, New York (2005)
10. Rhea, S., et al.: Handling churn in dhts. In: *USENIX Annual Technical Conference* (2004)
11. Dabek, F., et al.: Designing a dht for low latency and high throughput. In: *NSDI 2004*, pp. 85–98 (2004)
12. Krishnamurthy, S., et al.: A statistical theory of chord under churn. In: *PDP 2008*, pp. 473–482 (2008)
13. Marti, S., Ganesan, P., Garcia-Molina, H.: DHT routing using social links. In: Voelker, G.M., Shenker, S. (eds.) *IPTPS 2004. LNCS*, vol. 3279, pp. 100–111. Springer, Heidelberg (2005)

Selfish Neighbor Selection in Peer-to-Peer Backup and Storage Applications

Pietro Michiardi and Laszlo Toka

EURECOM

Abstract. In this work we tackle the problem of on-line backup with a peer-to-peer approach. In contrast to current peer-to-peer architectures that build upon distributed hash-tables, we investigate whether an un-coordinated approach to data placement would prove effective in providing embedded incentives for users to offer local resources to the system. By modeling peers as selfish entities striving for minimizing their cost in participating to the system, we analyze equilibrium topologies that materialize from the process of peer selection, whereby peers establish bi-lateral links that involve storing data in a symmetric way. System stratification, that is the emergence of clusters gathering peers with similar contribution efforts, is an essential outcome of the peer selection process: peers are lured to improve the “quality” of local resources they provide to access clusters with lower operational costs. Our results are corroborated by a numerical evaluation of the system that builds upon a polynomial-time best-response algorithm to the selfish neighbor selection game.

1 Introduction

During the last few years the on-line backup and storage market has witnessed an increasing interest from both academia and industry. Current commercial solutions and research projects present a variety of approaches to the problem of reliable, scalable and available on-line storage for heterogeneous users requiring to store and access a large amount of data from anywhere on the Internet [1,2,3]. While it is possible to draw a line to separate storage-centric (e.g. Amazon S3 [1]) from sharing-centric (e.g. Wuala [2], AllMyData [3]) approaches, the latter adding the possibility for users to operate fine-grained access control policies on their on-line data with a social-networking flavor, from an architectural point of view on-line backup and storage services can be broken down into those based on server farms [1] and those embracing the peer-to-peer (P2P) paradigm [2,3].

In this work we focus on P2P approaches, and study an architecture wherein peers are allowed to optimize the amount of resources they dedicate to the system. Specifically, the focus of this paper is on the *neighbor selection* algorithm, which is used by peers to decide where to place fractions of data they need to store. The lack of attention to neighbor selection is mainly due to the structured approach suggested by current system design, e.g. Wuala [2]. In a structured

approach (based on a Distributed Hash Table (DHT)), neighbor selection is *implicit* because data is uniformly stored on peers. The main benefit of DHT-based approaches is that they achieve load balancing by spreading data on every peer, irrespectively of their characteristics. Peer heterogeneity in terms of the amount of resources they dedicate to the system cannot be easily taken into account. As a consequence, such systems require additional layers to elicit users' cooperation to the system.

In this paper, we propose an unstructured architecture and study the implications of an uncoordinated neighbor selection algorithm wherein peers are responsible for building up their neighborhood which will store their data. Neighbor selection is modeled as a non-cooperative game in which users selfishly minimize the cost they bear for storing data. In our setting we introduce a global rank of peers in terms of their *profiles*, *i.e.*, the amount and quality of resources they offer to the system. We show that the neighbor selection process reaches an equilibrium in which the system is *stratified*: peers with similar profile cooperate by building bi-lateral links that are used to exchange and store data. The higher the peers' profiles are, the less costly the service they receive from one another is. The consequence of system stratification is a natural incentive for peers to improve the amount and quality of resources they offer to other peers.

The contributions of this paper are the following: *i*) in Sec. 3 we present a novel system design that has built-in incentives for peers to offer resources and to improve the quality thereof; *ii*) we define a simple model, in Sec. 4, that incorporates the cost for storing data in a selfish setting; *iii*) in Sec. 5 we design a polynomial-time algorithm to compute the equilibrium state of our system; *iv*) we show in Sec. 6, using a simulator, that the implication of peer selfishness is a stratified system in which peers can lower their cost of storing data.

2 Related Work

Reliable on-line backup and storage has been the subject of a very large number of prior works, both from the research community and from industry.

A notable example of current commercial solutions is represented by Amazon S3 [1], which is based on large clusters of commodity hardware running a custom-built distributed data structure discussed in [4]. The main goals of Amazon S3 are availability and reliability, which are achieved using replication. Availability and reliability do not come for free: end users are compelled to pay for the amount of space they occupy on the data centers and the amount of traffic their content generate [1].

The hybrid P2P design of some on-line backup and storage services such as Wuala [2] and AllMyData [3] require a centralized component to ensure a minimum storage space to end-users which is complemented by storage space at all peers taking part to the application. In current P2P systems availability and reliability are guaranteed by data redundancy through coding. For example, in Wuala data placement is achieved through a DHT layer, in which super-nodes are in charge of uniformly spreading the data on storage nodes. Incentives constitute

a key component of Wuala: users must offer an amount of local space inversely proportional to their on-line time [5]. Super-nodes are involved in constantly checking that this constraint is satisfied. Additionally, a distributed reputation mechanism serves the purpose of providing tit-for-tat incentives for users to allocate a large fraction of bandwidth to the P2P network. In contrast to these systems, our goal is to come up with an alternative system design that does not require external incentive mechanisms to support the system operation.

Research on distributed backup and storage applications has proliferated in the literature, although targeting different scenarios than the one we consider here. OceanStore [6], FarSite [7] and TotalRecall [8] represent influential design of such systems, the first based on a mesh of peers that cooperate in storing replicated (for active data) or redundant (for permanent data) blocks, the second using a randomized placement algorithm, the last using a DHT-based approach in selecting the placement of erasure coded data blocks. [9] provided insights to the performance of different data replication strategies in terms of data availability and durability.

Several prior works tackle the analysis of backup and storage applications, although from a more theoretic perspective. For example, [10] studied the potential benefits of a monopoly driven currency-based economy in P2P storage systems, and is orthogonal to this work. Among many other works that offer solutions to ensure fairness in the contributed and consumed storage, [11] suggested to create incentives to users by exploiting their social relationships.

Interestingly, the approach we take in this work can be seen as a network formation game [12], although we depart from the original mathematical tools used to analyze stable topologies.

3 System Design

Due to the uncoordinated nature of P2P backup applications, data availability, *i.e.*, ensuring that files can be retrieved in any moment, is an important issue that needs to be addressed. Similarly to related works such as Wuala, in our system we adopt data redundancy using erasure coding¹: files are split into c equally sized pieces which are then encoded to obtain n blocks. The original file can be reconstructed from any c fragments, where the combined size for the c fragments is approximately equal to the original file size. We term $k = n/c$ the redundancy factor of the coding scheme. File availability can be expressed as

$$\sigma = \sum_{i=c}^n \binom{n}{i} p^i (1-p)^{n-i} \quad (1)$$

where p is the *average* on-line time of peers that compose the system, and σ is the probability that the file is available. Given the average on-line time of peers and the number of fragments c that compose the original file, it is possible to

¹ See [8] for related works on replication and redundancy techniques to achieve data availability.

derive the redundancy factor k that meets the target file availability, which is achieved only if each of the n encoded blocks are placed on *distinct* peers. We explicitly derive the expression of k later in this Section.

Current P2P backup applications assume the average peer on-line availability p to be known and use a global redundancy factor for the whole system: once file fragments are encoded, they are spread uniformly at random on remote locations. *Asymmetric* data placement calls for complex mechanisms to enforce contribution of local space. Indeed, as opposed to the barter-based nature of exchanges we study in this work where *direct* retaliation is possible, the multi-lateral nature of asymmetric systems calls for auxiliary instruments, e.g. virtual currency, storage claims [13], to foster peer cooperation. Furthermore, the randomized nature of data placement also implies that the price of unreliable peers is shared among all system participants. Assume, for example, the on-line availability of peers to be distributed according to the normal distribution, that is $p \sim \mathcal{N}(\mu, \sigma^2)$. Then, the probability for a peer i whose availability is $p_i \gg \mu$ to store data on a peer j with availability $p_j \geq p_i$ is very small, and vice-versa. Hence, there is no reason for a peer to improve availability, and an additional mechanism compelling peers to offer more resources is required. We now define the resources playing an important role in P2P backup and storage applications.

Definition 1. *The resources peer i offers to the system are:*

- *storage space, $\check{c}_i \in \mathbb{N}$, that is the amount of encoded data chunks a peer stores locally for other peers;*
- *on-line availability, $p_i \in [0, 1]$, expressed as a probability for peer i to be found on-line;*
- *bandwidth $b_i = \min\{u_i, d_i\}$, where u_i , d_i represent respectively the upload and download capacity allocated by the user to the P2P application.*

These factors are tightly coupled: for example, a large amount of local space is useless when peer availability is low; similarly, high availability and space dedicated to the system operation are worth little if the bandwidth allocated to data exchange is not sufficient.

The endeavor of this work is to come up with a system architecture providing *embedded* incentives to foster peer cooperation without requiring any additional mechanisms. We advocate an *unstructured* P2P application with the following objectives: *i*) peers are compelled to offer a fraction of their local storage to other peers; *ii*) peers are incited to increase the on-line time and bandwidth they dedicate to the system. Intuitively, the first objective refers to the “quantity” of resources a peer offers while the second goal addresses the “quality” of such resources. In our system, *neighbor selection* replaces the inherent mapping of data chunks a peer stores in the system achieved by DHT-based solutions. As opposed to selecting remote storage locations uniformly at random, peers are left with the freedom of building a set of neighbors that will hold their data, and are not limited to their social acquaintances [11]. Formally, the problem can be described as follows.

Definition 2. Let \mathcal{I} denote the set of peers in the system, where $|\mathcal{I}| = N$. Every peer i splits their content in \hat{c}_i equally sized pieces. Pieces are encoded so as to obtain $n_i = k_i \hat{c}_i$ chunks.

Peers are responsible for establishing (logical) links to remote peers that will store their data, with the constraint that both ends of the link are required to agree to store data for each other: data placement is *symmetric*.

Definition 3. Let ν_i be the set of peers $\{j \mid j \in \nu_i \Leftrightarrow i \in \nu_j\}$, that is the link $i \leftrightarrow j$ is bi-directional. We call ν_i the neighbor set of peer i .

In our system, a peer is compelled to offer a fraction of local resources for the benefit of other peers. Because of the symmetric nature of our system design, peers are constrained to allocate an amount of storage space equal to the number of encoded chunks they would inject into the system. We can state the above constraint as follows:

Definition 4. A peer i that needs to store $n_i = k_i \hat{c}_i$ chunks in the system is required to offer an amount of local space equal to: $\check{c}_i = k_i \hat{c}_i \forall i \in \mathcal{I}$.

In practice, we have that $|\nu_i| = k_i \hat{c}_i$, otherwise either the backup data's availability drops due to the low number of peers based on Eq. 1 or unnecessary links are made if $|\nu_i| > k_i \hat{c}_i$. The cardinality of the neighbor set is increasing in the redundancy factor: the larger the redundancy employed in the coding scheme, the larger the number of *distinct* remote locations required to store data. Furthermore, we know that $k_i = f(p_j, \forall j \in \nu_i)$: according to Eq. 1, the redundancy factor is a decreasing function in the on-line time of remote peers that are part of the neighbor set of peer i .

Before proceeding any further, we extend the traditional definition of peer availability to account for the amount of bandwidth a peer dedicates to data exchanges: $\tilde{p}_i = p_i^{b_{ref}/b_i}$. The exponent modulates p_i by the fraction of bandwidth b_i peer i dedicates to the system compared to a reference value b_{ref} . b_{ref} is heuristically set to $\max(b_i)$ for $\forall i \in \mathcal{I}$. Hence, peer availability is slightly underestimated: the consequence for a peer is the requirement for a slightly larger neighbor set size that would compensate “slow” connections.

In this work we assume peers to be rational and selfish: intuitively, selfishness implies that peer will prefer to place data on remote peers offering resources of higher quality. Although this concept will be formalized in Sec. 4, we introduce peer selfishness in a simplified setting. Let ν_i and ν'_i be two distinct neighbor set peer i could link to, such as²: $\tilde{p}_j = p \forall j \in \nu_i$ and $\tilde{p}_k = p' \forall k \in \nu'_i$. Now, let's assume $p' < p$. It follows from our previous observations that:

$$k_i = f(\tilde{p}_j \in \nu_i) < k'_i = f(\tilde{p}_k \in \nu'_i) \Rightarrow \check{c}_i = k_i \hat{c}_i < \check{c}'_i = k'_i \hat{c}_i$$

² Instead of assuming all peers of ν_i to have the same availability p , it is possible to show similar results for the case in which $p = 1/|\nu_i| \sum_{j \in \nu_i} p_j$ or $p = \min_{j \in \nu_i} p_j$.

In words, selfish peers prefer to store data on remote peers with higher availability because this implies a reduced demand in terms of local storage space, following the constraint given in Def. 4. \square

Proposition 1. *In a symmetric system, in which peer i selfishly selects remote locations to store data we have that $k_i = f(\tilde{p}_j, j \in \nu_i) = f(\tilde{p}_i)$. That is, the redundancy factor that meets per file availability requirements can be computed as a function of peer i 's on-line availability \tilde{p}_i .*

Proof. We know by Def. 3 that $j \in \nu_i \Leftrightarrow i \in \nu_j$. Due to the selfish nature of peers and the symmetric nature of links between them, we know that $j \in \nu_i \Leftrightarrow \tilde{p}_j \geq \tilde{p}_i$ and $i \in \nu_j \Leftrightarrow \tilde{p}_i \geq \tilde{p}_j$. Hence, we have that $\tilde{p}_i = \tilde{p}_j$. The proposition follows directly. \square

With Prop. 1 at hand, we can formally define the redundancy factor k_i , which can be derived from Eq. 1 using the normal approximation to the binomial distribution:

$$k_i = \left(\frac{\sigma \sqrt{\frac{\tilde{p}_i(1-\tilde{p}_i)}{\tilde{c}_i}} + \sqrt{\frac{\sigma^2 \tilde{p}_i(1-\tilde{p}_i)}{\tilde{c}_i} + 4\tilde{p}_i}}{2\tilde{p}_i} \right)^2 \quad (2)$$

We now define peer *profiles*, which summarize the salient features of peers, as they compactly represent the “quality” of resources in terms of on-line time and bandwidth dedicated by a peer to the system. Profiles constitute a global ranking that is used during the execution of the neighbor selection mechanism discussed in Sec. 5. \square

Definition 5. *The profile of peer i is defined as follows: $\alpha_i = \frac{1}{k_i} \forall i \in \mathcal{I}$. We also define α_i^* to be the bootstrap profile of peer i when joining the system for the first time: $\alpha_i^* = \frac{1}{k_i^*}$ where $k_i^* = f(p_j, \forall j \in \nu^*)$ and ν^* is a random neighbor set.*

Before moving to a detailed description of selfish neighbor selection, we note that in this work we are making the implicit assumption that a method for monitoring the resources a peer dedicates to the system is available. The monitoring component would collect information on peer behavior in terms of profiles and truthful reporting on stored data. Depending on the application setting, the monitoring component can be centralized or distributed. It should be noted that in this paper we also gloss over data maintenance, which is an important problem as discussed in [14]. Due to space constraints, we cannot elaborate any further on monitoring and data maintenance and we will leave these aspects for an extended version of this work.

4 Peer Model

The central property of the system we investigate in this work is that peers are free to select the locations where their data chunks will be stored. Neighbor

selection is based on peer profiles: peers are assumed to be selfish in establishing links to remote peers holding high profiles and we are interested in studying the re-wiring process and its convergence properties. Here we describe the objective function peers optimize locally whereas in Sec. 5 we formalize the optimization framework that underlies our system.

The complex interplay between peers hinders the task of defining a peer model that accurately mimics the P2P system we investigate in this work. For this reason we define a simple heuristic cost function that incorporates the effects of peer selection and that accounts for the quality of resources offered by peers to the system.

Definition 6. *The cost C_i that peer i with profile α_i “pays” for storing $n_i = k_i \hat{c}_i$ units of data in a neighborhood ν_i is defined as follows:*

$$C_i = D_i(\alpha_i, \alpha_j \in \nu_i) + O_i(\alpha_i) + E_i(\alpha_i, \alpha_i^*) = \begin{cases} \log\left(\frac{\hat{c}_i}{\alpha_i^2} + \frac{1}{\alpha_i} + \left(\frac{\alpha_i}{\alpha_i^*}\right)^2\right) & \text{if } |\nu_i| \geq n_i \\ +\infty & \text{otherwise} \end{cases}$$

where the additive terms represent:

- *Degradation cost, D_i :* a target file availability can only be achieved if n_i units of data can be stored on $|\nu_i|$ *distinct* peers; hence this term indicates whether the selected file availability can be reached, that is when the neighbor set size $|\nu_i| \geq n_i$; if this is the case, the degradation cost decreases with the “quality” of peer i ’s neighborhood given by the profiles α_j of its members; hence, D_i is simply the aggregate profile of the neighbor set of peer i :

$$D_i = \sum_{j \in \nu_i} \frac{1}{\alpha_j} = \sum_{j \in \nu_i} k_j = |\nu_i| k_j \equiv k_i^2 \hat{c}_i = \frac{\hat{c}_i}{\alpha_i^2}$$

- *Opportunity cost, O_i :* describes the cost for peer i due to the loss of local storage space dedicated to hold data for other peers and it is inversely proportional to peer i ’s profile; by Def. 4, a high profile implies a small redundancy factor thus a smaller \hat{c}_i ; hence, O_i is simply the storage overhead compared to an ideally reliable system:

$$O_i = \frac{\tilde{c}_i}{\hat{c}_i} = k_i \equiv \frac{1}{\alpha_i}$$

- *Effort cost, E_i :* this term describes the cost induced by a variation in the “quality” of resources peer i offers to the system; the effort cost E_i is not trivial to derive: it follows from the non-linearity of Eq. 2 and the variation in the quality of resources peer i offers compared to the initial state.

We now build upon the user model defined in this Section and formalize the neighbor selection process.

5 Selfish Neighbor Selection

We study selfish neighbor selection using tools akin to non-cooperative game theory. First, we give a formal definition of the game, then we focus on the algorithmic nature of the optimization problem driving the neighbor selection process. As outlined in [15], selfish neighbor selection can be casted as a *stable exchange* (SE) game built on Def. 6 in which peer profiles constitute a (global) preference ordering. Indeed, the SE game belongs to the family of *matching* problems and, as in their simplest version (e.g. the stable marriage problem [16]), peers prefer links to remote peers holding a high profile.

Definition 7. *The stable exchange game is defined as follows:*

- \mathcal{I} denotes the player set (N is the number of players);
- \mathcal{S} is the strategy sets available to players: $\mathcal{S} = (\mathcal{S}_i)$ for $\forall i \in \mathcal{I}$; \mathcal{S}_i accounts for the combination of the two strategic variables: $\alpha_i \in [0, 1]$ and ν_i ;
- \mathcal{C}_i denotes the cost to player i on the combination of the strategy sets.

In the SE game every peer i seeks to minimize the cost function \mathcal{C}_i by setting appropriately the two strategic variables α_i and ν_i . Note that \mathcal{C}_i also depends on the strategic choice of other players j and that the creation of a link between two peers is conditioned to a bilateral agreement [17]. We now define the optimal strategy for a peer i and the Nash equilibrium of the SE game:

Definition 8. *The best response strategy for peer i , $s_i^* = (\alpha_i^*, \nu_i^*) \in \mathcal{S}_i$ is obtained by solving the equation $\arg \min_{\alpha_i, \nu_i} \mathcal{C}_i(s_i)$. In (Nash) equilibrium we have that $\mathcal{C}_i(s_i^*, s_{-i}^*) \leq \mathcal{C}_i(s_i', s_{-i}^*)$ for any player i and for any alternative strategy $s_i' \neq s_i^*$, where $s_{-i}^* = (\alpha_j, \nu_j)$ depicts the composition of equilibrium strategy of players other than i .*

Due to space limitations, the proof of the existence of the Nash equilibrium will be included in an extended version of this work.

Informally, we can interpret the SE game as follows: there are three forces that drive the decision process of player i , expressed in the function \mathcal{C}_i . The opportunity cost pushes player i to increase α_i because this implies a lower redundancy factor k_i hence a decreased amount of local storage offered to the system. The effort cost drives player i to a lower α_i , *i.e.*, reduced on-line probability p_i and allocated bandwidth b_i . The degradation term helps in balancing the two first opposing forces: depending on storage requirements \hat{c}_i , the profile α_j of other peers and the number of remote peers with $\alpha_j \geq \alpha_i$ that are eligible for a bilateral agreement, peer i could be better off increasing or decreasing α_i .

Neighbor selection brings to *system stratification*, which is the phenomenon we observe in our game. Our system stabilizes when peers are grouped into clusters, pooling users that have similar profiles³. A cluster of peers characterized by high profiles has lower operational costs than one with lower profiles. First, the

³ Analytical proofs of the existence, number and size of clusters is out of the scope of this paper and will be addressed in our future work.

redundancy factor used by peers in a high-profile cluster is small compared to a low-profile cluster, hence peers will have to dedicate a smaller amount of local space to other peers. Second, reliable peers store data on similarly reliable peers while unreliable peers are bound to store data on other unreliable peers, and are compelled to improve the quality of resources they dedicate to the system as the number of unreliable peers shrinks.

We now describe how we implement in an efficient way the *iterated best-response* [18] algorithm to the SE game in order to find an equilibrium. We split the optimization problem that player i faces regarding the strategic variables α_i and ν_i : profile selection and neighborhood construction are *interleaved*. The profile selection is implemented using a technique based on the simulated annealing method [19]: in each iteration of the best-response algorithm, player i randomly modifies α_i by a discrete, fixed value and *estimate* the alteration of the cost \mathcal{C}_i due to the change. A decreasing function $f_k(\mathcal{C}_i^k)$ of the total cost at iteration k of the algorithm is then used to decide whether the new value of α should be adopted or discarded. Once profiles are set, the algorithm solves an instance of the stable matching problem using an extension [15] of the Irving's algorithm to the stable fixtures problem [20]. This procedure is repeated until an equilibrium is found. In [15] we show that the iterated best response algorithm described above runs in *polynomial time*.

It is important to note that, in practice, only once the process of neighbor selection reaches a steady state for all peers, that is, no peer has an incentive to re-wire to other remote peers, the actual data transfer will take place. We also emphasize that our model accounts for a static system set during the neighborhood selection process. Targeting the bootstrap issue of a real growing system is on our agenda.

6 Numerical Evaluation

In this section we focus on a numerical evaluation of the neighbor selection process. We built a synchronous simulator, in which time is slotted, and implemented the iterated best-response algorithm discussed in Sec. 5. We examine a closed system in which $|\mathcal{I}| = N = 100$ and bootstrap profiles are normally distributed with mean $\alpha = 0.55$. We assume $\hat{c}_i = 10 \forall i \in \mathcal{I}$.

Our goal is to examine the properties of the equilibrium of the SE game (labeled *strategic*) and to compare equilibrium solutions to a simulated DHT-based system (labeled *random*). The implicit hypothesis, to make the two cases comparable, is that peers evaluate their costs similarly. Our evaluation is based on the following metrics:

- Total cost: $\mathcal{C}(t) = \sum_{i \in \mathcal{I}} \mathcal{C}_i(t)$, which cumulates the cost that every peer has to cover for storing their data in the system;
- Average user profile: $\alpha(t) = 1/N \sum_{i \in \mathcal{I}} \alpha_i(t)$, which is the average profile computed at each round of the iterated best-response algorithm;
- Cumulative Distribution Function (CDF) of equilibrium user profiles;

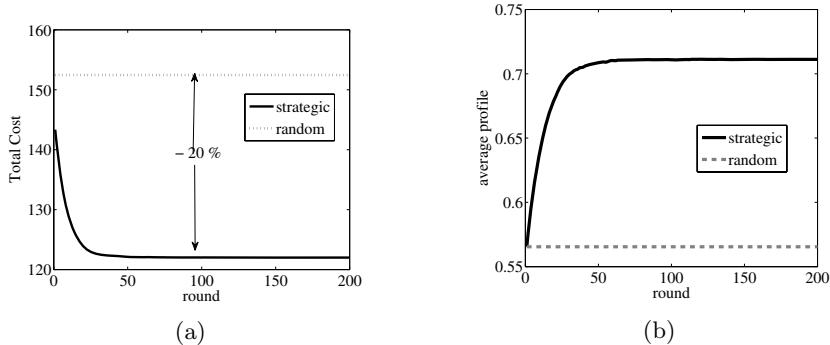


Fig. 1. Time-series of the aggregate cost $C(t)$ and the average user profiles $\alpha(t)$ for the random and strategic neighbor selection

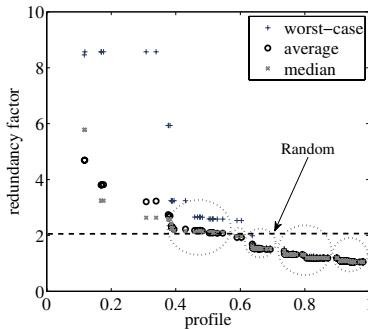


Fig. 2. Redundancy factor adopted by each peer (sorted by their profile) for random and strategic neighbor selection

- Profile improvement: $\Delta_i = \alpha_i - \alpha_i^*$ for $\forall i \in \mathcal{I}$;
- Redundancy factor: $k_i \forall i \in \mathcal{I}$. Since clusters may not be exactly uniform in terms of user profiles we report worst, mean and median values;

Due to the randomized nature of our algorithm, the results presented in the following are averaged over 10 simulation runs. In the following, the legend “round” stands for the iteration number of the best-response algorithm.

Fig. II(a) illustrates the improvement of strategic neighbor selection when compared to the random neighbor selection: by rewiring their connections according to the iterated best-response to the stable exchange game, peers are able to reduce their costs, and the aggregate figure decreases by 20%. The underlying reason for reduced operational costs is shown in Fig. II(b). Starting from the same random bootstrap profiles α_i^* , strategic peers increase their profiles while in the random case peer profiles do not change in time and remain fixed to bootstrap values.

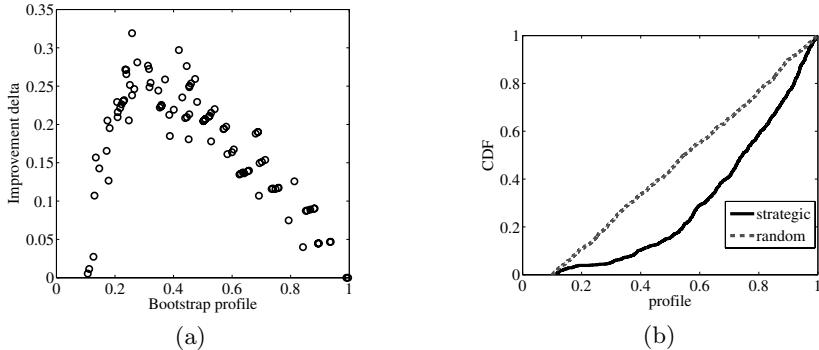


Fig. 3. Improvement and distribution of equilibrium profiles

Fig. 2 illustrates the redundancy factor adopted by each peer in the system, when using the random or strategic neighbor selection policy. In the random case every peer uses the same redundancy factor: due to the bootstrap profile distribution, the median and mean redundancy factors coincide, and sum to roughly 2 (indicated in the figure with a dashed horizontal line). In the worst case, the redundancy factor is 10 (which corresponds to the y-axis limit). Instead, the strategic neighbor selection differentiates peers in clusters. Peers with a high profile (close to 1) use a lower redundancy factor than peers belonging to a cluster with lower profile (closer to 0). We enriched Fig. 2 to illustrate the clustering phenomenon that emerges at the end of the rewiring process. Peer clusters are emphasized with dotted circles around groups of peers holding *similar* profile. On the upper-left corner of the figure we notice the presence of outliers: for these peers, the effort cost becomes predominant, hence their redundancy factor is very high.

Fig. 3(a) shows the difference between the bootstrap and the equilibrium profiles. We observe that peers holding “extreme” profiles (either high or low values) have less incentives to improve their ranking. We notice a maximum improvement (which amounts to almost 35% difference) for peers with a profile slightly less than the average profile. The exact value of the maximum improvement depends on the input setting to the stable exchange game. Finally, Fig. 3(b) shows that the majority of peers apply a substantial improvement to their profiles as compared to the initial profile distribution.

7 Conclusion and Future Work

Armed with the realization that current P2P backup and storage applications require complex mechanisms to foster peer cooperation in this paper we presented an alternative system design in which peers, as opposed to previous works, are left with the choice of selecting locations to store data. We introduced a distinction in the amount and the quality of resources peers contribute to the system

and showed that selfish neighbor selection alone contributes to the key feature of our approach: incentives to share local resources and to improve their quality are embedded in the system design. In this paper we modeled data placement as a game in which peers minimize the cost for storing data in the system; we also gave a polynomial-time algorithm to compute the equilibrium of the system.

We simulated the selfish neighbor selection process and showed that the system converges to a stratified state: peers are clustered based on their contributions and storage costs are inversely proportional to clusters' quality. This result represented the key motivation for a peer to improve the quality of resources dedicated to other peers.

The results presented in this paper open paths for several future directions: our ultimate goal being the real implementation of such a system we will focus on the analysis of the convergence properties of selfish neighbor selection in an asynchronous setting and on a distributed implementation. We will also focus on the evaluation of system overhead, both in terms of monitoring and repair activities.

References

1. Amazon S3, <http://aws.amazon.com>
2. Wuala, <http://wua.la>
3. AllMyData, <http://allmydata.org>
4. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. In: ACM/USENIX SOSP (2007)
5. Grolimund, D., Meisser, L., Schmid, S., Wattenhofer, R.: Havelaar: A Robust and Efficient Reputation System for Active Peer-to-Peer Systems. In: NetEcon (2006)
6. Kubiatowicz, J., Bindel, D., Chen, Y., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., Zhao, B.: Oceanstore: An architecture for global-scale persistent storage. In: ACM ASPLOS (2000)
7. Adya, A., Bolosky, W., Castro, M., Cermak, G., Chaiken, R., Douceur, J., Howell, J., Lorch, J., Theimer, M., Wattenhofer, R.: FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In: USENIX OSDI (2002)
8. Bhagwan, R., Tati, K., Cheng, Y., Savage, S., Voelker, G.M.: TotalRecall: System Support for Automated Availability Management. In: ACM/USENIX NSDI (2004)
9. Chun, B., Dabek, F., Haeberlen, A., Sit, E., Weatherspoon, H., Kaashoek, M.F., Kubiatowicz, J., Morris, R.: Efficient Replica Maintenance for Distributed Storage Systems. In: ACM/USENIX NSDI (2006)
10. Maille, P., Toka, L.: Managing a Peer-to-Peer Data Storage System in a Selfish Society. In: IEEE JSAC (2008)
11. Li, J., Dabek, F.: F2F: reliable storage in open networks. In: IPTPS (2006)
12. Fabrikant, A., Luthra, A., Maneva, E., Papadimitriou, C.H., Shenker, S.: On a Network Creation Game. In: ACM PODC (2003)
13. Cox, L.P., Noble, B.D.: Samsara: Honor Among Thieves in Peer-to-Peer Storage. In: ACM/USENIX SOSP (2003)
14. Duminuco, A., Biersack, E.W., En-Najjary, T.: Proactive replication in distributed storage systems using machine availability estimation. In: ACM CONEXT (2007)

15. Toka, L., Michiardi, P.: A dynamic exchange game. In: ACM PODC (2008)
16. Gale, D., Shapley, L.S.: College Admissions and the Stability of Marriage. *American Mathematical Monthly* 69 (1962)
17. Corbo, J., Parkes, D.C.: The price of selfish behavior in bilateral network formation. In: ACM PODC (2005)
18. Fudenberg, D., Tirole, J.: Game Theory. MIT Press, Cambridge (1991)
19. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by Simulated Annealing. *Science* 220(4598) (1983)
20. Irving, R.W., Scott, S.: The stable fixtures problem - A many-to-many extension of stable roommates. *Discrete Applied Mathematics* 155(17) (2007)

Zero-Day Reconciliation of BitTorrent Users with Their ISPs

Marco Slot¹, Paolo Costa^{1,2}, Guillaume Pierre¹, and Vivek Rai¹

¹ VU University Amsterdam

² Microsoft Research Cambridge

Abstract. BitTorrent users and consumer ISPs are often pictured as having opposite interests, with end-users aggressively trying to improve their download times, while ISPs throttle this traffic to reduce their costs. However, inefficiencies in both download time and quantity of long-distance traffic originate in BitTorrent randomly selecting peers to interact with. We show that biasing the link selection allows one to reduce both median download times by up to 32% and long-distance traffic by up to 16%. This optimization can be deployed by modifying only the BitTorrent trackers. No external infrastructure nor specialized client-side software deployment is necessary, thereby facilitating the adoption of our technique.

1 Introduction

Peer-to-peer applications have significantly changed the landscape of Internet traffic management. While traditional client-server applications used to generate a majority of localized download traffic, peer-to-peer applications generate large amounts of global outgoing traffic. The impact is such that some fear the Internet will run into serious capacity problems within a few years [1]. In particular, BitTorrent, being reported as the foremost contributor of Internet traffic, has created a new antagonism between end users and their ISPs. While BitTorrent implementations deploy aggressive data transfer strategies to reduce file download times, consumer ISPs are forced to buy transit from global networks, driving up their operational costs. As a result some ISPs use throttling strategies to keep their costs under control [2], which in turn impacts the download times of their customers.

The reason why BitTorrent generates so much global traffic is that each peer of a given torrent selects other peers to exchange data with in a random fashion, without any consideration of network distance. Each peer then continuously updates its active connection set in a greedy fashion in favor of peers that can provide the best upload rates. The relative worse performance of long-distance peers may somewhat induce BitTorrent clients to exchange data with peers located nearby, but only among those in their local peer set.

This paper shows that careful selection of the initial peer sets given to each BitTorrent client can significantly reduce both the user-perceived download times and the generated amount of long-distance traffic. While this idea is not new, all

existing implementations rely on the global deployment of either network measurement infrastructures [3,4] or client-side extensions [5,6]. We argue that none of these approaches are practical, since they both require massive deployment of specialized software before beneficial effects become noticeable for the users and their ISPs. For example, while the authors of the Ono plug-in for Azureus can legitimately be proud of having deployed their code to over 400,000 clients, such a number represents only a small fraction of all BitTorrent users and therefore has little impact on global Internet traffic.

We propose to bias the connections maintained by BitTorrent clients towards nearby nodes using coordinate-based latency prediction. Importantly, our approach requires no global software adoption or deployment. The only required operation to optimize a whole torrent is to update its tracker with our software, and install a handful of globally distributed “landmark” servers. In our experiments we run a dozen of landmarks in PlanetLab, but in a real setting we expect that multiple BitTorrent trackers would organize themselves to become landmarks for each other. We refer to such reconciliation of both users and ISPs interests as *zero-day*, in reference to zero-day security attacks which can be launched at any time by mere exploitation of already deployed software.

Our approach relies on passive latency measurement. Peers are made to open TCP connections with each landmark by adding the addresses of landmarks to the first list of peers returned by the tracker. The measurements obtained from the TCP connection are used to compute the location of the peer in a centralized network coordinate system (GNP) [7]. The tracker can thus reply to any subsequent request from that peer with a list of carefully selected peers in place of the usual random choice. Applying this selection bias significantly reduces traffic cost for ISPs, while reducing download times for the tracker’s end-users. We evaluate our approach through carefully designed simulations, which we validate against PlanetLab. We show that it allows to reduce the median download times by up to 32%, and the quantity of global Tier-1 traffic generated by up to 16%.

2 Related Work

Several approaches proposed to bias BitTorrent traffic towards nearby peers. However, they all rely on large external infrastructures or client-side modifications, which largely hampers their applicability.

Bindal *et al.* proposed bias based on peers’ ISPs [5]. A peer selects k peers within the same ISP and the rest from other ISPs. Unfortunately this requires either modified BitTorrent clients or specialized infrastructure at the ISPs which makes deployment difficult. Moreover, the whole approach depends on a high number of peers sharing the same ISP, which rarely happens in practice. Finally, this approach does not distinguish between two ISPs in the same town, which are likely to have some peering relationships in place, and two ISPs in different continents, which, instead, must acquire transit from Tier-1 ISPs in order to communicate.

The iPlane project monitors routes from hundreds of locations to build a single, coherent view of the Internet [4]. iPlane can help build a biased BitTorrent tracker. Rather than using only network distance, it also uses loss rates to estimate the TCP throughput between peers. The fine-grained network measurements and models of iPlane may produce more accurate results than GNP, but they require a huge infrastructure. This infrastructure is currently deployed on PlanetLab for research purposes, but deploying a commercial or public alternative would be much more expensive than a cooperative GNP system. Finally, the iPlane topology maps are very large (several GBytes) and look-ups computationally expensive.

P4P allows ISPs to publish their network information and preference for peers to connect with [3]. This solution exploits a network oracle akin to iPlane but here the ISPs can provide fine-grain network information. P4P has support from a few large ISPs, but to be effective it would need to be supported by a vast majority of ISPs worldwide and in collaboration with BitTorrent trackers.

The Azureus BitTorrent client implements the Vivaldi network coordinate system [8]. Vivaldi strongly resembles GNP, but uses peer-to-peer interaction instead of fixed landmarks. Although this approach could allow Azureus clients to bias the choice of peers to unchoke, this technique is limited to Azureus clients only.

Some public databases (e.g., <http://www.ripe.net>) enable to map IP addresses to ISPs, thus possibly removing the need of tracking peer's location through landmarks. This information, is however often incomplete and coarse-grained. Similarly to [5], it does not allow to identify nearby peers from different ISPs.

Finally, Ono is a plug-in for the Azureus BitTorrent client that relies on Akamai [6]. Akamai owns servers in many locations, and redirects its clients to the closest replica through DNS. Each Ono peer resolves DNS names of popular websites hosted by Akamai. If two peers receive the same IP addresses, they are likely to be relatively co-located. Ono uses this information to select nearby peers. Ono's applicability is however unclear. Although it has been downloaded 400,000 times, this represents only a small fraction of all BitTorrent users. Hence, any gains made due to selection will only have local effects. In addition, the authors show that Ono-enabled clients experienced slightly *lower* median download rates than regular ones.

3 Background

3.1 BitTorrent

BitTorrent is a peer-to-peer protocol for distributing large files [9]. Each file offered for distribution uses a separate overlay, managed by one tracker responsible for maintaining the overlay membership. To join a BitTorrent network, a client registers at the corresponding tracker, with the identity of the torrent to join and its own contact address¹. The tracker adds the new peer to the membership

¹ Although some recent versions of BitTorrent clients also support a DHT-based decentralized tracker, most available torrents normally rely on a centralized tracker [10].

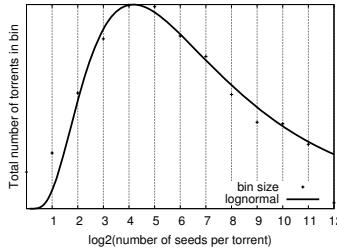


Fig. 1. Swarm size distribution (number of peers in swarms with $2^{x-1} < \text{size} \leq 2^x$)

list, and returns a random subset of this list to the client to build its initial *peer set*.

BitTorrent relies on incentives to encourage participants to contribute their upload bandwidth in the file distribution process. The incentive scheme is based on reciprocation such that a peer uploads content only to those from which it receives something in return. Each peer autonomously decides which peers to *unchoke*, that is which peers to send file content to. Similarly, each peer can decide to *choke* a peer that does not upload sufficient data and unchoke another peer in the hope to obtain a better throughput. BitTorrent thus uses only local greedy policies in selecting the destination of uploaded traffic so as to maximize its own download rate. Other possible optimization metrics such as the implied cost of long-distance traffic are not taken into consideration. The local peer set is refreshed only when peers disconnect due to client exit, connection failure or long period of inactivity. This makes the initial peer selection particularly critical to provide the peer with a good set of nodes to choose from.

Importantly, any form of a peer selection bias can be effective only when applied to large enough networks. If the swarm is smaller than the peer set size (usually 50), clients will have all peers in their peer set. We studied the distribution of network sizes that any peer may be part of, by screen-scraping the number of peers from 150,000 torrents on a popular BitTorrent tracker. Figure 1 shows that about half of the clients belong to a network of 82 peers or more, and are therefore likely to benefit from biased link selection. In addition, we can expect the networks they are in to generate the majority of traffic as many smaller networks are idle.

3.2 ISP Economics

ISPs use two types of relationship to carry traffic between machines located in different ISPs. First, two ISPs that exchange large amount of traffic may *peer* with each other, that is establish a direct connection between their two networks. The cost of a peering relationship is virtually constant regardless of the quantity of traffic exchanged. Other traffic is sent through a *Tier-1* ISP, whose business is to carry traffic between ISPs. Tier-1 ISPs charge their service on a per-volume

and sometimes per-distance basis. Consumer ISPs therefore have a financial incentive to reduce the volume of traffic that their users upload into Tier-1 ISPs.

BitTorrent, by uploading large quantities of traffic to randomly selected nodes creates a financial burden to consumer ISPs. Since peering can only be done between ISPs that are physically close, most of this traffic must be sent over Tier-1 networks. Some ISPs have reportedly tried to reduce their costs by throttling BitTorrent traffic. Although this may indeed reduce their peak traffic rate, it creates bad customer experience. A better solution in our opinion is to deploy mechanisms such that the majority of BitTorrent traffic is exchanged inside ISPs or through peering relationships instead of Tier-1 ISPs. As we show later, such measures have the favorable side effect of reducing user-perceived BitTorrent download times.

3.3 Peer-to-Peer Connection Throughput

Download time improvements derive from the fact that reducing the latency of paths has the side-effect of improving the connection throughput. We consider three main parameters that together contribute to defining the throughput in BitTorrent.

Access link capacity. Many users access the Internet through asymmetric cable or ADSL connections. This has important implications for BitTorrent where each downloaded packet is in principle reciprocated with one uploaded packet: download link capacities are rarely saturated by BitTorrent traffic. The only bandwidth bottleneck concerns the upload capacity. Even though this limitation remains constant regardless of the BitTorrent node selection strategy, upload capacities are not necessarily utilized to their full extent in real scenarios, and potentially provide room for download time improvement.

Internet throughput. Although the core Internet links can be considered as well provisioned compared to the end users access links, there exists a weak, inverse correlation between latency and bandwidth [11, 12]. Therefore, reducing the latency of paths used for BitTorrent traffic can likely increase the throughput of these connections. Similarly, shorter paths are likely to exhibit lower packet loss rates.

TCP throughput. Any single TCP connection has a maximum achievable throughput driven by inter-node latency, and regardless of the capacity of the underlying network. This bottleneck is due to the fact that TCP window sizes are limited to 65536 bytes. In the total absence of packet loss, the maximum throughput of a single connection is $\frac{W_{max}}{RTT}$, where W_{max} is the maximum window size and RTT is the round-trip time. Any packet loss further reduces this maximum throughput.

4 Design and Implementation

Our approach relies on instrumenting the BitTorrent tracker so that it can estimate inter-peer network latencies, and return biased selections of links to each

peer. Each time a peer P connects to the tracker, the tracker returns a set of other peers which exhibit low latency paths to P . The client remains completely unaware of this bias and behaves as usual. However, as we show in the next section, due to the low latency of these paths, these connections are more efficient than in the traditional BitTorrent and enable saving download time and reducing inter-AS hops.

To build this, one should address three questions: (i) the tracker must estimate inter-peer latencies, although issuing N^2 measurements in a torrent with N peers is unacceptable; (ii) latency measurements must be realized with no explicit support from the peers themselves to preserve our client-independent approach; (iii) the tracker must select links to return to each peer so as to favor low latencies without compromising other important properties of the BitTorrent swarm.

4.1 Latency Estimation

Directly measuring the pairwise latencies between peers of a given torrent would require $O(N^2)$ measurements, which is unacceptable for large torrents. A classical solution to this problem is the use of GNP “network coordinates” where each node is given a d -dimensional coordinate and inter-node latency is estimated as the Euclidean distance between coordinates [7,13]. The advantage of coordinate-based latency prediction is that it only requires latency measurements from each node to $d + 1$ landmarks, as opposed to N^2 for pairwise measurements.

In the initial phase, each landmark measures its round-trip-time to every other landmark. Landmark coordinates can then be computed such that the Euclidean distance between coordinates matches the measured latencies. This translates into a minimization problem over an error function ε . The procedure to determine the coordinates for a regular node is similar to the initial phase. The round-trip-time between the node and each landmark is measured and coordinates are computed by minimizing the sum of the error function. GNP can predict over 90% of latencies within 50% relative error. This is generally sufficient to find nearby hosts.

4.2 Passive Latency Measurements

Each time a new peer joins a torrent, we need to issue latency measurements between the peer and each of the landmarks to derive the new peer’s coordinate. However, we cannot expect any explicit support from the peer itself since we do not want to rely on specialized software at the client side. Instead, we rely exclusively on the BitTorrent protocol itself and on passive latency measurements.

Whenever a new BitTorrent client registers at the tracker, the tracker returns a small set of randomly selected peers (like a normal tracker), plus the addresses of the landmarks. Since clients start with an empty peer set and do not distinguish between actual peers and the landmarks they will open connections to each. A landmark can then passively measure its latency to the peer by measuring delays between packets during the TCP three-way handshake [14]. Once a connection is established, the landmark gracefully closes the connection and reports the measured latency to the tracker. When the tracker has received

enough measurements from the landmarks, it can compute the peers' coordinate. The client, after several unsuccessful connections to landmarks, will contact the tracker again to obtain a fresh set of peers. The tracker can then predict the latency between the client and other peers using the Euclidean distance between the nodes' coordinates.

While a tracker may decide to run landmarks itself, we expect small groups of trackers to organize and provide each other with landmarks for their mutual benefit. The associated workload is very low. Each client must be measured once by each landmark, and the implied network overhead is very low. Trackers can also maintain a memory of past measurements, which further reduces the overhead due to latency measurements [13].

4.3 Peer Selection

When a BitTorrent client contacts the tracker for new peers, the tracker should not systematically return the n closest peers to this client. First, doing this creates a risk of partitioning the swarm into disjoint cliques. Second, it would reduce the interest for a client to ask for new peer sets, since these sets would remain largely identical from one request to another.

Our modified tracker returns a number of peers selected randomly from the 25% closest peers, plus a few more peers selected randomly in the whole swarm. The first measure increases the gain of repeatedly contacting the tracker, while the second keeps the swarm connected. We however note that in our experiments we found it extremely difficult to partition BitTorrent swarms, even when the tracker returns no long-distance link. We discuss this further in Section 5.5.

Another potential issue is that, when a new torrent is created, the tracker initially has only few nodes that can be returned to the clients. If the tracker would return a full peer set, this would create a clique of nodes all connected to each other. Any subsequent node that joins the torrent would find it difficult to connect to any pre-existing node. We prefer returning a smaller number of links to the first clients that show up, so that subsequent nodes can join more easily.

5 Evaluation

Evaluating a BitTorrent optimization in a realistic setting is very challenging due to the difficulty of involving hundreds of users across the world. We evaluate our system on PlanetLab, although the large bandwidths between PlanetLab nodes are not representative of most BitTorrent users. We therefore also developed a simulator that reproduces the network conditions experienced by regular BitTorrent clients, communicating over the Internet. The simulator models propagation delay, TCP throughput and upload capacity sharing. We feed it using data obtained from actual BitTorrent clients and use PlanetLab experiments to validate the simulator².

² All our implementations can be found at <http://marcoslot.net/latorrent.htm>

The simulator implements the standard BitTorrent algorithms [15][16], including choking, optimistic unchoking, strict piece priority, request pipelining and rarest first piece selection. Messages are delayed based on latency data and TCP throughput is estimated after the popular PFTK model [17] with a window size of 2^{16} bytes, taking into account fair sharing of the access link capacity.

5.1 Experimental Settings

We first present a set of experiments using real measurements taken from PlanetLab. We focus here on validating our simulator by comparing the results obtained in both approaches. We deployed the original BitTorrent client (version 5.2.0) on 141 PlanetLab nodes, and our landmark implementation on 7 nodes. We fed our simulator with the same configuration using measured latencies and packet loss rates between these nodes.

We tested our system under two scenarios with maximum sizes of 200 and 1,000 nodes to analyze the impact of biased selection in small and large-scale BitTorrent networks. To extract a representative set of peers to use in our simulations we used the data provided by iPlane [4], a service providing accurate loss, latency and path predictions for a large number of Internet hosts. iPlane periodically participates in BitTorrent swarms to also measure access link bandwidths. From the resulting set of BitTorrent peers and their pairwise latency, loss and path predictions we used uniformly drawn samples in the simulations.

To reproduce a realistic join rate, both in the PlanetLab- and iPlane-based experiments, we took segments out of the Izal tracker log [18]. We introduce new peers based on the offset of the starting times in the log and continue doing so until we reach the maximum number of peers. The peers download a 256MB file originating from a single seed, which remains available during the whole experiment. The seed has above-average upload capacity. Since our completion times will not match the tracker log we do not use the tracker departure times. Instead we let peers stay with mean departure time of 2 minutes after completing their download.

We measure the following metrics: (i) the *download time*, defined as the time required for a peer to download the whole file; (ii) the *latency* of network paths, weighted by the quantity of traffic actually exchanged via each path; and (iii) the fraction of traffic exchanged through a *Tier-1* ISP. We considered ASNs 174, 209, 701, 1239, 1299, 1668, 2914, 3356, 3549, 3561, 6453 and 7018 as Tier-1 ISPs.

Each experiment was repeated 5 times in each configuration.

5.2 Simulator Validation

To validate the accuracy of our simulator on a real wide-area network, we run our system in PlanetLab and compared the results against those obtained from the simulator, fed with the real values of latency, bandwidth and packet-loss rate as measured on PlanetLab. Table II shows that our simulator is indeed successful

Table 1. PlanetLab vs. Simulated performance

	Download time			Latency		
	PlanetLab	Simulated	$\Delta(\%)$	PlanetLab	Simulated	$\Delta(\%)$
Standard(median)	357 s	347 s	2.08%	57 ms	49 ms	14.0%
Biased (median)	286 s	288 s	0.7%	55 ms	45 ms	18.2%
Standard (90-th percentile)	1700 s	1704 s	0.2%	268 ms	254 ms	5.2%
Biased (90-th percentile)	1251 s	1387 s	9.2%	267 ms	240 ms	10.1%

(a) Download time and latency

	PlanetLab	Simulated	$\Delta(\%)$
Standard	41%	39%	4.8%
Biased	39%	38%	2.5%

(b) Tier-1 Traffic

at reproducing the behavior of a real network, providing a good approximation of the real performance for all three metrics introduced above³.

This is a key result for two reasons. First, it allows us to concentrate on simulations to evaluate the effectiveness of our approach in settings typical of real BitTorrent swarms. Second, it shows that our biased tracker reduces the median download rate by 20% and the 90-th percentile by 26%. Conversely, the improvement is lower in terms of Tier-1 traffic. The reason is that most traffic in PlanetLab is routed through few different university networks, so the incidence of Tier-1 traffic is relatively limited. As we will observe later in this section, in realistic settings the Tier-1 traffic accounts for more than 80% of the whole traffic.

5.3 Simulation Results

We now turn our attention to the evaluation of our approach based on simulations, and using a more realistic set of nodes and network metrics taken from iPlane.

Tier-1 traffic. Our goal is to reduce both the cost of ISPs and the user download times. To analyze the cost of ISPs we determine how much traffic is routed through a Tier-1 ISP. Table 2 confirms that a large fraction of standard BitTorrent traffic is routed through Tier-1 ISPs, and therefore results in a direct financial cost for consumer ISPs. However, using the biased tracker the Tier-1 traffic is reduced by 11% for a network of 200 peers and by 16% for a network of 1000 peers.

Note that these results have been obtained using a uniformly random sample of nodes from the iPlane dataset. This means that nodes are uniformly spread around the globe, therefore making Tier-1 traffic unavoidable. However, many torrents are only of regional interest [19]. We expect that our approach would

³ Note that even though the *relative* value of the median error for the latency is rather high ($\sim 15\%$), its *absolute* value ($< 10\text{ms}$) makes it practically negligible.

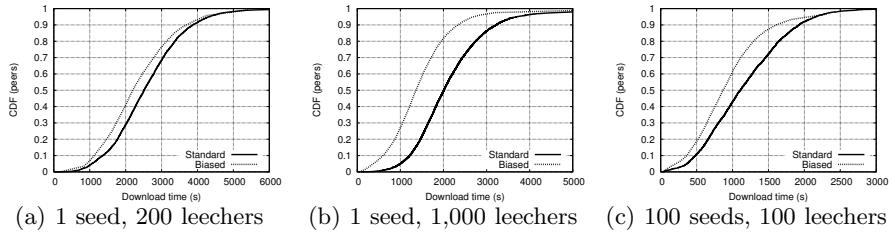


Fig. 2. Number of peers with download time $< x$

Table 2. Percentage of traffic routed through Tier-1 ISPs

Network size	Standard	Biased	Gain(%)
200 nodes	81%	73%	11%
1,000 nodes	81%	68%	16%

perform better in such settings, since the tracker would have more short paths to choose from.

Download Time. Figure 2 reports the CDF of download times for a 200-node and a 1000-node network. The biased approach delivers much better download times, with improvement of the median download time of respectively 12% and 32%. Large networks perform better because the tracker then has more nodes to choose from.

These two graphs represent a worst-case scenario in the sense that the swarm has only a single seed. However, real torrents normally have several seeds available at any time. In the data from Figure 1 we observed on average 5 seeds for every 3 leechers. Figure 2(c) shows a 200-nodes scenario where the swarm contains 100 seeds and 100 leechers. The median gain in download time raises from 12% to 22%.

Traffic Latency. To estimate the distance over which traffic is sent, Figure 3 shows the CDF of latency experienced by traffic. Results show a large decrease of latency when using the biased tracker, with median improvements of respectively 33% and 75%. Again, the large-scale network exhibits a greater improvement.

5.4 Adaptive Sample Size

In previous experiments, the tracker always returned a full peer sample size (50 peers). However, as discussed in Section 4.3, this can degrade the performance because early nodes may establish links to distant nodes and later prevent closer nodes to contact them. We now evaluate our system when the size of the sample returned is $2 \times \sqrt{N}$, where N is the current swarm size.

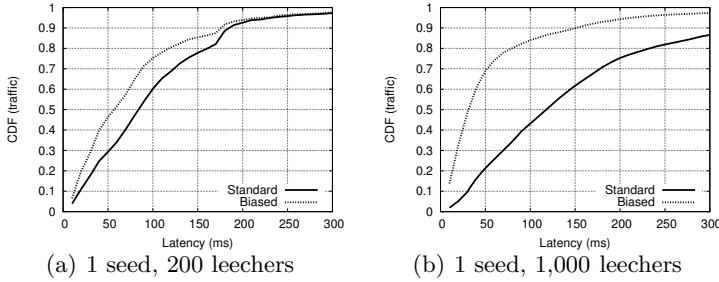


Fig. 3. Traffic exchanged over links with latency $< x$

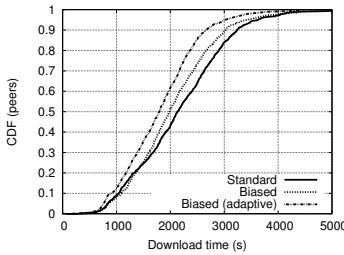


Fig. 4. Number of peers with download time $< x$ (1 seed, 200 leechers)

Figure 4 evaluates the adaptive strategy against the standard and biased ones in the 200-leechers scenario. To better isolate the phenomenon we focus only on the first 200 peers which completed the download. Clearly, performance benefits from the adaptive solution. While the biased tracker achieves a gain of 12% compared to the standard tracker, adapting the sample size to the current size of the network gains 18%. Even an unbiased tracker shows a small gain using this technique.

5.5 Bias Degree

As detailed in Section 4.3, our biased tracker returns a few random nodes (10% in our experiments) to prevent network partitions. Figure 5 shows the effect of varying the bias degree in the 200-leechers scenario, ranging from a fully random selection ($bias=0$) to a fully biased one ($bias=1$). Even with a full bias no network partitioning appears, and our system performs even better. The reason is that the default size of the peer set (50 nodes) is large enough to ensure strong connectivity of the entire network. On the other hand, for lower values of the peer set, we can foresee that partitions may occur. Investigating the impact of bias using different values of the peer sets is part of our immediate research agenda.

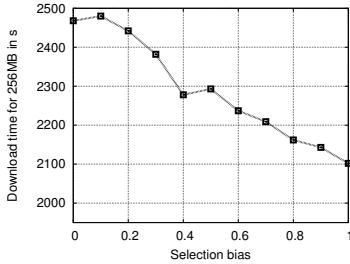


Fig. 5. Median download times for different bias degrees

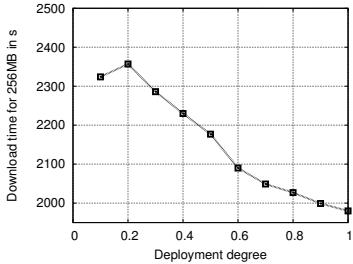


Fig. 6. Median download times for a client-based implementation against different deployment degrees

5.6 Deployment Degree

We claim that modifying only the tracker allows to optimize entire swarms at a time, while client-based approaches depend on the number of clients which implement the latency-based bias. Figure 6 shows the performance of an Ono-like approach [6] for different fractions of Ono-enabled clients in the swarm. It is difficult for us to reproduce the accuracy of Ono’s latency estimations. Instead, we assume here that enabled clients have a *perfect* latency predictor among themselves to select nearby links. We observe that, despite the very favorable assumptions taken, in order to achieve comparable performance to our approach, more than 50% of clients must participate. In real settings, with inaccurate latency prediction, this number would reasonably be even higher. Given the vast number of BitTorrent implementations, this would be very hard to achieve in practice.

6 Conclusions

BitTorrent encourages peers to donate their unused networking resources to help distribute popular content. The tremendous success of this simple idea however sharply increased the amount of global traffic uploaded by end users. We showed in this paper that relatively simple optimizations can provide significant performance improvements, with 11-16% reduction of expensive Tier-1 traffic in challenging scenarios. We also find the median download times are reduced by 12-32%, providing strong incentive to deploy these optimizations.

We explicitly aimed at easy deployment of our approach. We therefore excluded any solution that would require code deployment at the client side, or the prior availability of massive global services. Instead, optimizing a whole torrent requires only a modified tracker and a handful of landmarks. We even expect that the need for explicit landmark deployment will disappear if multiple trackers agree to cooperate and behave as each other’s landmarks.

We present this work and make our implementations freely available in the hope to attract the attention of major BitTorrent tracker operators. Should they adopt this technique, the seemingly opposite interests of BitTorrent users and their ISPs might be partially reconciled.

Acknowledgments

We wish to thank Harsha Madhyastha for providing access to the iPlane data [4] and Michal Szymaniak for his SYNACK/ACK and GNP implementations [13].

References

1. CNET: AT&T: Internet to hit full capacity by 2010, http://news.cnet.com/ATT-Internet-to-hit-full-capacity-by-2010/2100-1034_3-6237715.html
2. CNET: Comcast on the hot seat over BitTorrent, http://news.cnet.com/Comcast-on-the-hot-seat-over-BitTorrent/2009-1025_3-6231975.html
3. Xie, H., Yang, Y.R., Krishnamurthy, A., Liu, Y.G., Silberschatz, A.: P4p: provider portal for applications. In: Proc. SIGCOMM (2008)
4. Madhyastha, H.V., Isdal, T., Piatek, M., Dixon, C., Anderson, T.E., Krishnamurthy, A., Venkataramani, A.: iPlane: An Information Plane for Distributed Services. In: Proc. OSDI (2006)
5. Bindal, R., Cao, P., Chan, W., Medved, J., Suwala, G., Bates, T., Zhang, A.: Improving Traffic Locality in BitTorrent via Biased Neighbor Selection. In: Proc. ICDCS (2006)
6. Choffnes, D.R., Bustamante, F.E.: Taming the Torrent: A practical approach to reducing cross-ISP traffic in P2P systems. In: Proc. SIGCOMM (2008)
7. Ng, T.S.E., Zhang, H.: Predicting Internet Network Distance with Coordinates-Based Approaches. In: Proc. INFOCOM (2002)
8. Dabek, F., Cox, R., Kaashoek, M.F., Morris, R.: Vivaldi: a decentralized network coordinate system. In: Proc. SIGCOMM (2004)
9. Cohen, B.: Incentives Build Robustness in BitTorrent. In: Proc. First Workshop on Economics of Peer-to-Peer Systems (2003)
10. Crosby, S.A., Wallach, D.S.: An Analysis of BitTorrent's Two Kademlia-Based DHTs. Technical Report TR-07-04, Rice University (2007)
11. Oppenheimer, D., Chun, B., Patterson, D., Snoeren, A.C., Vahdat, A.: Service placement in a shared wide-area platform. In: Proc. USENIX Technical Conf. (2006)
12. Lee, S.J., Sharma, P., Banerjee, S., Basu, S., Fonseca, R.: Measuring Bandwidth Between PlanetLab Nodes. In: Proc. 6th Intl. Workshop on Passive and Active Network Measurement (2005)
13. Szymaniak, M., Presotto, D., Pierre, G., van Steen, M.: Practical large-scale latency estimation. Elsevier Computer Networks 52(7) (2008)
14. Jiang, H., Dovrolis, C.: Passive estimation of TCP round-trip times. In: Proc. SIGCOMM (2002)
15. bittorrent.org: BitTorrent specification, http://www.bittorrent.org/beps/bep_0003.html
16. theory.org: Unofficial BitTorrent specification, <http://wiki.theory.org/BitTorrentSpecification>
17. Padhye, J., Firoiu, V., Towsley, D.F., Kurose, J.F.: Modeling TCP throughput: A simple model and its empirical validation. In: Proc. SIGCOMM (1998)
18. Izal, M., Urvoy-Keller, G., Biersack, E.W., Felber, P., Hamra, A.A., Garcés-Erice, L.: Dissecting BitTorrent: Five Months in a Torrent's Lifetime. In: Proc. 5th Intl. Workshop on Passive and Active Network Measurement (2004)
19. Iosup, A., Garbacki, P., Pouwelse, J.A., Epema, D.H.J.: Three lessons from one peer-level view. In: Proc. 11th Conf. of the Advanced School for Computing and Imaging (2005)

Surfing Peer-to-Peer IPTV: Distributed Channel Switching

A.-M. Kermarrec¹, E. Le Merrer^{1, *}, Y. Liu^{2,3, *}, and G. Simon²

¹ INRIA Centre Rennes - Bretagne Atlantique, France

² Institut TELECOM - TELECOM Bretagne, France

³ State Key Laboratory of Networking and Switching Technology,
Beijing University of Posts and Telecommunications, China

Abstract. It is now common for IPTV systems attracting millions of users to be based on a peer-to-peer (P2P) architecture. In such systems, each channel is typically associated with one P2P overlay network connecting the users. This significantly enhances the user experience by relieving the source from dealing with all connections. Yet, the joining process resulting in a peer to be integrated in channel overlay usually requires a significant amount of time. As a consequence, switching from one channel to another is far to be as fast as in IPTV solutions provided by telco operators. In this paper, we tackle the issue of efficient channel switching in P2P IPTV system. This is to the best of our knowledge the first study on this topic. First, we conducted and analyzed a set of measurements of one of the most popular P2P systems (PPLive). These measurements reveal that the set of contacts that a joining peer receives from the central server are of the utmost importance in the start-up process. On those neighbors, depends the speed to acquire the first video frames to play. We then formulate the switching problem, and propose a simple distributed algorithm, as an illustration of the concept, which aims at leveraging the presence of peers in the network to fasten the switch process. The principle is that each peer maintains as neighbors peers involved in other channels, providing peers with *good* contacts upon channel switching. Finally, simulations show that our approach leads to substantial improvements on the channel switching time. As our algorithmic solution does not have any prerequisite on the overlays, it appears to be an appealing add-on for existing P2P IPTV systems.

1 Introduction

The diffusion of television over Internet, known as IPTV, has fostered a huge amount of works. Among the most studied architectures, *P2P systems* have produced not only theoretical proposals (see [16] for an overview of the main theoretical challenges) but also practical applications used by millions of users (*e.g.*, PPLive, sopcast, PPStream). For instance, the number of visitors of PPLive [1] website reached 50 millions for the opening celebration of Olympics (source:

* Supported by project P2Pim@ges, of the French Media & Networks cluster.

Data Center of China Internet) while the dedicated Olympic channel attracted 221 million of users in only two weeks. However, these implementations do not provide efficient channel switching features, while this is well-known as a natural TV watcher behavior. Typically, the time interval from when one new channel is selected until actual playback starts on the screen can be prohibitively long in current P2P streaming systems. More specifically, the only work that has, to the best of our knowledge, evaluated this start-up delay reports that it requires from 10 to 20 seconds to switch to a popular channel and up to 2 minutes for less popular channels [10]. Should these recent measurements highlighting the high frequency of switching system be confirmed [5,6], this could be a burden for the success of P2P IPTV systems.

In multicast-based IPTV (networks controlled by telco operators), a set of solutions has been designed to reduce the start-up delay [7] upon channel switching. They mostly consist in sending data of some *adjacent* channels along with the current channel data. Two channels C_1 and C_2 are called adjacent if when a user watches C_1 , if (s)he switches channels, the probability that C_2 is chosen is high. For example, it has been shown that a user watching a sport channel, has a high probability to switch to another sport channel [5]. Thus, should the user switch from a channel to one in the adjacent channel set, the corresponding multicast traffic is received without suffering from any network delay. Several works have extended this technique in order to maximize the probability that the target channels are within the set of adjacent channels [13,3].

Yet, in P2P IPTV systems, receiving simultaneously several multimedia flows, even degraded, remains too expensive (important overhead of applicative multicast over IP, compared to lower layer multicast). However, users switching patterns being similar [5], we leverage the fact that switching channels mostly involve adjacent channels. In this paper, we make an initial step to analyze and solve the channel switching issue in P2P IPTV with a simple approach that could be potentially implemented over all existing P2P systems. The other works related with multi-channel systems, concurrent to this study, have not addressed the neighborhood discovery problem [18,19]. Our contributions are threefold.

First, in an attempt to characterize the criticity of joining a new channel for the playback delay, we measure and analyze the PP live system [1] focusing on the so-called *bootstrap time*: the time between the reception of the P2P contacts, *i.e.*, a peerlist (a list of supposedly active peers given by a central server for a given channel) and the time at which the first video packet is received. A joining peer is expected to be able to discover new peers from this initial contact lists through request propagation in its new channel overlay. However, our measures show that in PP live, most of the initial video content is actually provided by those contacts given by the server. Our study demonstrates their importance as well as shows that the ratio of peers effectively active in the peerlist is particularly low.

Second, we define the distributed channel switching problem and describe a simple yet efficient solution in which a peer watching a given channel also keeps some links to few peers in specific channels, typically adjacent channels. The peers with which a peer exchanges video content related with its channel are

called *overlay neighbors*. The peers maintained from other channels are called *contact peers*. When a peer x has a contact peer in the channel c , x is a *switcher* for c . Obviously a peer cannot be a switcher for all channels as the traffic generated for the maintaining of contact peers can not be neglected. Instead, peers may leverage their overlay neighbors to find switchers for more channels. Our goal is to ensure that, in a given overlay neighborhood, the number of adjacent channels covered by switchers is maximized. We show that an implementation solution of the switching responsibility distribution over a given overlay network is closely related with a (r, k) -*configuration problem* [8], a NP-hard problem. From a theoretical side, no exact solution can be computed in a reasonable time, even if one could have a global view of the system. Therefore, we provide a practical solution approximating the optimal solution. Our algorithm is simple, local, efficient and able to cope with dynamic behavior of P2P systems.

Although, this algorithm has been designed for channel switching, its applicability goes beyond. More generally, the problem addressed in this paper consists of switching from a highly connected clusters of peers to another highly connected clusters within a giant overlay network. Typically, switching from one chapter to another chapter in a P2P VoD streaming system admits a very close problem formulation, and solutions are unsurprisingly related with prefetching of most probable seeking positions [20, 9].

Finally, we provide a comprehensive set of simulations. Actually, it is difficult to compare our proposal to other existing implemented solutions because, to the best of our knowledge, only centralized algorithms are used by current systems. Yet, we show that our proposal significantly improves the quality of peers that are given to a joining peer in a channel, and then in practice reduces time for this peer to get the first video packets (start-up delay reduction).

Totally, the main contributions are the following: (i) we measure and analyze the bootstrapping of PP live, motivating the need for a faster process; (ii) we formulate the problem of distributed channel switching, and (iii) we propose and simulate a greedy algorithm to the distributed channel switching problem.

The rest of this paper is organized as follows. Section 2 details our measurements of start-up delays of PP live. Section 3 formulates the channel switching problem. Section 4 presents our algorithm proposal. Section 5 reports simulation results, and Section 6 finally concludes the paper.

2 On the Importance of Given Peer Sets in PP live

To understand the bootstrap process, and assess its importance for current popular P2P streaming systems, we conducted a measurement study of PP live in July and August 2008. Application protocols are not publicized by PP live, justifying a reverse-engineering by practical measurements (*active crawling* and *passive sniffing* [10]), to understand this critical phase. We focus on the bootstrapping process, *i.e.* the first two minutes of connections. Depending on channel popularity estimated by the PP live website, we define five classes of channels: from *1-star* popularity grade (the less popular channels) to *5-star* popularity grade. In most

Channel popularity	% of responding peerlist's peers	Avg number of overlay neighbors	% of ov. neighbors \notin peerlist	T_{start} (seconds)	(sec- onds)	T_{start} \notin peerlist
1	2.1	4.90	34.69	9.61	25.44	
2	7.0	16.40	35.98	10.73	34.49	
3	10.9	23.20	32.44	11.92	49.84	
4	17.2	33.75	23.70	11.21	63.27	
5	16.1	30.87	21.86	9.76	47.46	

Fig. 1. Measurement results for bootstrap procedure in PPLive

of experiments (more than 95%), the residential peer receives the *peerlists* from three servers. Each peerlist consists of 50 peers that are assumed to watch the same channel, in order for the joining peer to bootstrap. 20 channels are selected per class, except for the 5-star class for which 8 channels only were available.

A first set of results is reported in Fig. 1 (first 4 columns). We focus here on overlay neighbors providing at least one video packet during the first two minutes. These peers are known either through the initial peerlist, or through subsequent requests from joining peers to their neighbors. We observe that the ratio of peers that belong to the initial peerlist and actually send eventually a packet (second column) is low. This shows the low quality of peers provided through the bootstrap process. Less than 17% of peers that have been provided by servers for the bootstrap are delivering video content. For less popular channels, this ratio is even worse (2.1%). This can be explained by the fact that peers are active during a smaller duration in non popular channels, so servers' knowledge of active peers is unperfect. Presence of NATs, and overhead tradeoff for refreshing of peer information towards the servers may explain these low ratios. This is an issue that motivates the need for a dedicated strategy for improving the discovery of trustworthy peers for the bootstrap process. It can be done by contacting less peers, but preferably good and active ones for efficiency. Then, we enumerate the number of peers that deliver at least one packet during the first two minutes (third column); we observe that more peers are forwarding content in most popular channels, thus virtually providing an increased quality of service. Finally, the fourth column shows that approximately one third of peers that are actively providing content (protocol neighbors) are not given by servers in peerlists. Those results show that the initial peerlists given to a joining peer are crucial. If they are not good enough, an additional process for requesting new peers to participate is needed, therefore adding some delay for video start. Note that this need is even more stringent for low popularity channels.

A second set of results is depicted in Fig. 1 (last two columns). We measure the time T_{start} which is the average interval between getting the initial peerlist from servers and receiving the first packet. In all cases, the first packet is received from a peer in the initial peerlist. The results show that peers that are contacted are not immediately responding, and the required time to send the packet is approximately constant, around 10 seconds. The last column shows the time at which the first packet is received from a peer obtained by an additional search process (not in the initial peerlist). The results are very different depending on

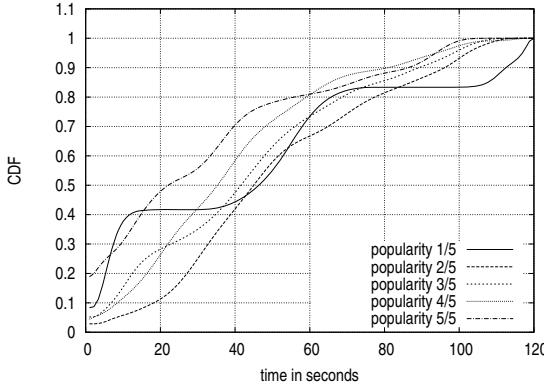


Fig. 2. CDF of the duration of video packet delivering for transient neighbors

the popularity of channels (time increases with popularity). This shows that the additional process is more critical for unpopular channels. Effectively, [10] shows that playback starts up to 2 minutes after a join for unpopular channels; here the first packet is received at best 25 seconds after the reception of peerlists, highlighting this process criticity. Contrariwise, for popular channels, first packets arrive after playback started; they help for improvement of quality of service instead of initial playback.

The last measurement is depicted in Fig. 2. We present the time interval between the first packet and the last packet received from a peer. As we measure only the first two minutes of bootstrap, peers still sending packets after this delay are not shown here. Fig. 2 shows the results under a Cumulative Distribution Function (CDF) where a point at (0.4,45) means that 40% of transient peers have sent video packets during less than 45 seconds. This measurement shows how long a peer is expected to be transient in delivery to send video packets. Obviously, the longer that time, the easiest for the joining peer is to predict the filling of its video buffer. The main observation is that the curves are almost linear. Therefore, there is absolutely no guarantee on the expected video delivery duration for transient peers (time during which a peer will send video packets can not be predicted). The only curve we can analyze more easily is for the most unpopular channel: its reveals three typical durations about 10 seconds for 40% of peers, then about 50 seconds for 40% of peers and almost two minutes for others. As the video delivery for this channel is mostly ensured by peers provided through the additional process, we may argue those peers have a behavior that can be more easily predicted (probably due to a particular choice, according to their importance for the application quality).

3 The Distributed Switching Problem and Our Proposal

As previously shown, basic centralized managements of the bootstrap procedure, in particular the delivery of peerlists, may lead to a long playback delay.

Meanwhile, it has been shown in current mesh-based streaming systems [4] that some characteristics such as upload capacity for example may have a great impact on the quality of content dissemination. Therefore there is a tendency in such systems, to connect peers having close characteristics (upload capacity, latency, activity, *etc.*). We call such peers *matching peers* in the sequel.

In order for an overlay to connect peers regarding such characteristics, gossip-based *topology management systems* (see *e.g.*, [1]) consist in ensuring that all peers are eventually connected to matching peers, through an epidemic and iterative protocol. With a randomly chosen peerlist, a joining peer has low chance to be connected directly to matching peers, and thus should quickly improve neighborhood. Actually, despite some recent works in this direction (*e.g.*, [18]), a central server can hardly provide peers with matching peers quickly, in a scalable fashion. This calls for a new strategy. We describe our approach as follows.

Multi-channel IPTV Modelization. We assume an IPTV system consisting of α channels, each channel being modeled as an overlay graph. The set of all overlay graphs is noted \mathbb{G} . A peer x is involved in exactly one P2P channel $g(x) \in \mathbb{G}$ among the α channels simultaneously offered by the IPTV service. In the P2P system, x cooperates with some other peers, called *overlay neighbors*, with the exclusive goal of exchanging content related to the channel $g(x)$. This small set of neighbors is noted $\Gamma^{prot}(x)$. Note that we focus in this paper on the topology problem, and do not address the higher level policies of packet exchanges between peers. Furthermore, let $d(x, y)$ denote the hop-distance of the shortest path between a peer x and another peer y in the overlay graph ($d(x, y) = \infty$ if $g(x) \neq g(y)$). For any integer $k > 0$, let the *k-neighborhood* of x be $(\Gamma^{prot}(x))_k = \{y | 0 < d(x, y) \leq k\}$, where y is called *k-neighbor* of x .

Distributed Channel Switching Definition. We define the distributed channel switching problem has, for any peer, (*i*) the overlay change from a channel to another one, on a fully distributed fashion (*i.e.* without request to a central entity), and (*ii*) the quick matching with accurate peers in the new overlay, in order to shorten playback delay. These are the two metrics we promote and evaluate in this paper.

Our Proposal. Considering the fact that the number of available channels is not likely to grow indefinitely in practice, and that channels have an unequal distribution of popularity in reality (Pareto in [5]), approaches that may want to create a *structured overlay* (*distributed hash table, skip list*) connecting peers in all channels does not seem justified at the moment. Instead, we propose a light and simple mechanism, that aims at giving access to the most probable channels, thus capturing distributedly a fair amount of potential switches. To do so, besides the neighborhood that is used directly for the aim of the P2P protocol, a peer also maintains connections to peers belonging to other overlays, for channel switching

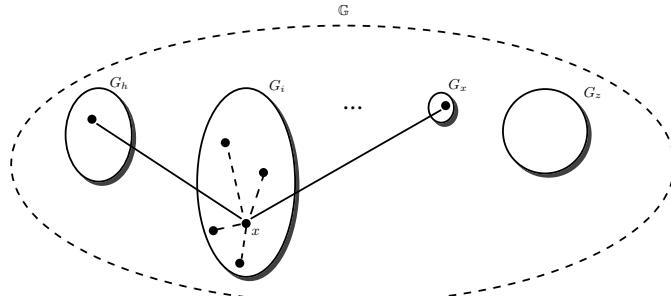


Fig. 3. An example of a resulting system \mathbb{G} . Node x has 4 neighbors in its current overlay G_i (dashed lines) and 2 contacts in other overlays, $\delta = 2$ (plain lines).

purposes. We note $\Gamma^{inter}(x)^c$ the set of contact peers of x , for the channel c . Formally, a peer y is a contact peer of x means that y is not in the same overlay ($g(y) \neq g(x)$), but $x \rightarrow y$ relation exists. A peer x is asked to maintain fresh lists of contacts (dead nodes or nodes that switched are removed from contact lists), and to try to match with those contacts.

We associate with a peer x a set $\mathcal{C}(x)$ of overlays in \mathbb{G} such that an overlay graph G belongs to $\mathcal{C}(x)$ if and only if there exists a peer y such that both $g(y) = G$ and $y \in \Gamma^{inter}(x)^G$. We say that x is a *switcher* to a channel if the overlay graph associated with this channel is in $\mathcal{C}(x)$. Because the number of channels α is expected to be large, a peer can not be a switcher to all other channels, for scalability issues (we bound by δ this maximal number). This forms a P2P overlay switching system depicted in Fig. 3.

System In a Nutshell. An *overlay switch* for a node x is the process leading to a complete change of its neighborhood $\Gamma^{prot}(x)$, to another one reflecting its move to a new chosen channel in \mathbb{G} . Say that a peer x in G_i wants to switch to a channel G_j ; we describe now the two ways to switch overlays.

First, a peer x , has a switcher peer y (thus y has a $\Gamma^{inter}(y)^{G_j}$) in its k -neighborhood. Search could be implemented through basic expanding ring search technique, at k hops, from the requesting peer. It then only has to ask y for its contact list from G_j , to replace its protocol neighborhood, and join the targeted channel. Note that, if both overlays G_i and G_j are based on a same matching preference system (say upload capacity), this is immediately leveraged in the new overlay (as few handshakes are necessary until x discovers matching neighbors in G_j) and will speed up the convergence process.

The second scenario is simply a failure in searching distributedly a contact peer for the targeted overlay. It can either occur because of the dynamics of the overlays (switchers are not accessible at k hops), or because the chosen channel is not in the most probable ones. In such a case, the traditional centralized approach is used to get a set of contacts. Note that in IPTV, servers are mandatory, as they are in charge of pushing the content into overlays.

Main Challenges

Allocating Contact Peers. In IPTV systems, some channels are far more popular than others [515]. Moreover, channels exhibit content similarities such that a participant is more likely to switch to an overlay having similar content or dealing with close activities as the one it is currently enjoying. That is, from one channel, the probability to switch to another channel is not equal for all channels. For example, paper [5] shows that 76% of switches are done in the same channel genre (*e.g.* sport, music or news). We would like every overlay to contain switchers having few contact peers in most probable channels, so that a peer that wants to switch is likely to find a switcher in its k -neighborhood.

Ensuring Matching Contact Peers. As we have seen through measurements in PPlive, the initial peerlist given to a joining peer is crucial for a quick start-up. We use the matching property in an overlay to directly give matching (or at least close) peers in the target overlay. As previously said, our assumption is that the neighborhood reflects the matching, in other words, for two integers k_1 and k_2 with $k_1 < k_2$, if y_1 is a k_1 -neighbor of x and y_2 is a k_2 -neighbor of x , then it means that y_1 matches better with x than y_2 . The challenge is here to maintain an accurate matching between peers, despite the overlay dynamics.

4 Protocol Description

This section depicts implementation of our proposal, summarized on Algorithm 1

Switcher Creation. In order for nodes to distributedly choose for which channels they should be switchers, we refer to the notion of *domination*, well known in graph theory. A set $D \subseteq V$ of vertices in a graph $G = (V, E)$ is called a *dominating set* if, for every vertex x in V , x is either an element of D or is adjacent to an element of D . More generally, a k -*dominating set* extends this adjacency notion at k hops, at most, from a given peer x [8]. Consider that at most δ resources (here channels to be switcher for) can be allocated to a node. A δ -*configuration* is an allocation of a set of resources such that, for every resource, the vertices associated with this resource form a dominating set. The same extension as in previous definitions can state for a δ -configuration. That is, a (δ, k) -*configuration* is to allocate resources to vertices, such that no more than δ different resources are allocated to any vertex, and each vertex can access a resource associated with another vertex in less than k hops. Back to our switcher creation problem, nodes in the dominating set are responsible for keeping contacts in specific channels, and the resources are the overlays one peer has to keep in touch with.

The purpose of (δ, k) -configuration is to determine a configuration; unfortunately this decision problem has been proved to be NP-complete. That is, all optimization problems that are directly related with this decision problems are NP-hard. Thus, maximizing the number of channels that can be allocated, with a given δ and a given k is NP-hard, as well as minimizing the maximal distance and the number of switcher peers δ for a given k and a given most

Algorithm 1. A simple protocol for probabilistic switching

Initially: upon IPTV join of a node x in channel c_i

- 1: Arbitrarily fixed input: k (depth of switcher search), t (awaited T-Man convergence for $\Gamma^{prot}(x)$)
- 2: Request server c_i that returns a peerlist $\Gamma^{prot}(x)$

Matching for x in current channel:

- 3: T-Man resulting in matched $\Gamma^{prot}(x)$ +
- 4: After t cycles, periodically search for switchers at k hops: +
- 5: Request switchers in $(\Gamma^{prot}(x))_k$
- 6: Pick a channel c_j in missing most probable, or choose the furthest one
- 7: Transform x into a switcher for c_j :
- 8: Request server c_j that returns a peer list $\Gamma^{inter}(x)^{C_j}$
- 9: T-Man resulting in a matched $\Gamma^{inter}(x)$

Upon switch to channel c_j :

- 10: Find a switcher y for c_j among $(\Gamma^{prot}(x))_\kappa$ with increasing κ ($\kappa \leq k$)
- 11: $\Gamma^{inter}(y)^{C_j}$ becomes $\Gamma^{prot}(x)$ in c_j
- 12: Else, request server c_j that returns a peerlist $\Gamma^{prot}(x)$
- 13: Goto line 3

probable channels to cover. Therefore, an optimal solution can not reasonably be computed in a dynamic large-scale system. Instead, we propose a heuristic which enables to provide a practical and realistic alternative.

We assume that each peer joining a channel is provided by the central server or by neighbors with the list of most probable channels for switching in the same channel genre. It then checks at k hops from itself if a switcher is missing, sorted by order of importance; if so, it becomes a switcher for this channel (an initial peerlist is acquired from the server), with a limit of δ channels. If switchers to all most probable channels have been found, the furthest ones are chosen (l. 4-6).

Matching through Gossip. In order for peers to get connected to matching peers, according to some application predefined metric (as *e.g.* proximity, latency or bandwidth), we use a gossip based topology management similar to T-Man [11]. In this paradigm, each peer owns a value reflecting this metric. To end up with neighbors close from this value (l. 3 & 9), each peer periodically chooses its neighbor with the closest value, and exchanges with it a list of current closest neighbors and some nodes chosen randomly amongst the channel population. After each exchange, closest nodes are kept as neighbors, while furthest ones are discarded. It turns out that only a few cycles are needed to reach a near optimal peer matching, even in presence of churn [11]. The random peers needed by this protocol are provided by peer sampling protocols, that can also be based on the gossip paradigm (see *e.g.* Cyclon [17] or the Peer Sampling Service [12]).

The same matching technique as for overlay neighbors Γ^{prot} is used by switchers for contacts they keep in touch with in Γ^{inter} , except that the procedure there is not bidirectional (the switcher tries to match with peers in target overlay, but the reverse case is not true). With this matching process, once a peer wants to

switch, and is able to find a switcher in its k -neighborhood, the peer contacts given by the switcher have relatively a high chance to be close from its future position in the new channel (l. 10-11).

5 Multi-channel System Simulation

We evaluated our proposal using the PeerSim [2] simulator targeting large-scale and dynamic overlays. The multi-channel system we implemented as an input is based on recent application measurements conducted in [5].

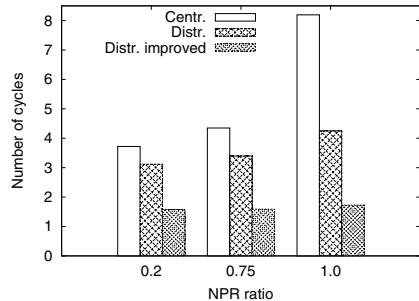
Simulation Configuration. The system is composed of 150 channels, classified in 13 genres (*e.g.* cine, sports, kids, docu). Channel popularity in each genre follows a Zipf distribution, that decays fast for non-popular channels; the probability to stay in the same genre during a switch is set to 76% [56]. Otherwise, a peer picks uniformly at random first the genre it switches to (probabilities are exposed in [5]), and then the channel in that genre. Furthermore, according to the time peers watch channels, peers get assigned a role: *surfer* (watches the channel from 0 to 45 seconds), *viewer* (46 to 3600) or *leaver* (3601 to 36000); joining and departing peers create the dynamism (or *churn*) in the system. Note that a second corresponds to one Peersim cycle execution in our simulation. After each channel switch, a peer picks a role with probability 0.6 for surfer, 0.35 for viewer and 0.05 for leaver.

Simulation parameters are set as follows: 10^5 peers in the system, an average channel-neighborhood Γ^{prot} per peer of 15 (consistent with observations made in Section 2). A peer could be a switcher for simply 1 channel ($\delta = 1$), and keeps in touch with 5 peers in this channel for Γ^{inter} (note that PPlive returns not less than three peerlists of 50 peers each). The gossip protocols (matching and sampling), for both Γ^{prot} and Γ^{inter} are executed at each cycle. After a switch, a peer waits for $t = 3$ cycles before looking for switchers, and this action is performed again every 3 cycles (to prevent a flood for search at every cycle).

Simulation Results. We run simulation 86400 cycles for an equivalent of one day of simulation. Key statistics, collected from 100 randomly selected peers, are presented on Fig. 4. Our implementation leads to 58% of distributed switches. Note that this percentage cannot be greater to the probability of switching in the same genre (here 76%), as we only provide switchers for channels in the same genre group (difference between 58% and 76% is due to churn and to the fact that only the 5 most probable channels are linked in each genre, and not all of them). The average distance between a switching peer and a switcher to the target overlay was 0.67 hops. This is due to the fact that a majority of peers are switchers themselves or could find one among their 1-neighbors.

The neighborhood perfection ratio (NPR) is a simple measure for the correctness of the matching of peers in our simulator: for a peer x , it is the number

	Centralized	Distributed
# switches	2718 (42%)	3760 (58%)
Switcher dist.	/	0.67 hops
Peerlist with ≥ 1 match	5.8%	33.3%
NPR	20%	73%

Fig. 4. Switching statistics for 100 peers**Fig. 5.** Convergence to matched neighbors

of best matches for x that are currently its neighbors, divided by the number of neighbors of x (1 then expresses that all neighbors of x are its best matches, and no better one exists in the overlay). We now observe the number of matching peers that are given to a switching peer, by the server or by our system. In the centralized case, random selection provides only 5.8% of peerlists containing at least one matching neighbor (20% of matched contacts on average). When switches are executed distributedly, one third of the provided peerlists contain at least one good contact (with a high value of NPR of 73%), thus highlighting the improvement in providing neighbors with close characteristics.

We now look (Fig. 5) at the number of cycles needed, after a switch, to reach a given NPR. As expected, the distributed process in every case helps switching peers to reach more quickly a given ratio, as the given peerlist is closer from the switching peer than random list given by the server. We also observe that centralized switches perform relatively well: this is due to the efficiency of gossip-based topology management which converges quickly in many scenarios [11]. This motivates current streaming applications to consider gossip-based protocols for the efficiency of neighborhood management. Finally, as an important part of switching peers (33%) is immediately provided with a high quality peerlist with respect to the matching criterion (NPR of 73%), we also plot convergence time for those peers (called "Distr. improved" on Fig. 5). We observe that the time required to reach a perfect neighborhood is very short. In this case, as nearly all given neighbors are matching, the missing ones could be found in very few cycles.

This simulation study shows that a simple implementation can improve in a substantial number of cases (i) the quality of the initial sets provided to switching peers, thus allowing (ii) a fast convergence towards matching neighborhoods. Finally (iii), gossip-based mechanisms appear to be an elegant and reliable solution to tackle this self-organization issue. Increased values for parameters δ , k and the number of channels linked in same genres obviously improve results. Based on observations from Section 2, this could potentially reduce significantly video start-up delays at the application level.

6 Conclusion

In this paper, we address the problem of switching from one channel to another in P2P IPTV systems, in a distributed fashion. To the best of our knowledge, no previous studies have dealt with scalable channel switching, which is a crucial issue for the next generation of multimedia delivery mechanisms over Internet. This approach shows the interest of leveraging peers' belonging to an overlay, in order to improve forthcoming switches. We believe that the simple proposed algorithm represent a first step toward the design of distributed and efficient switching mechanisms.

Acknowledgments

We would like to thank Dehao Zhang for the measurements of PPiive.

References

1. <http://www.pplive.com>
2. <http://peersim.sourceforge.net>
3. Boong Lee, D., Joo, H., Song, H.: An effective channel control algorithm for integrated iptv services over docsis catv networks. *IEEE Transactions on Broadcasting* 53, 789–796 (2007)
4. Boudani, A., Chen, Y., Simon, G.: A quicker way to discover nearby peers. In: Proc. of the ACM CoNEXT Conference (2007)
5. Cha, M., Rodriguez, P., Crowcroft, J., Moon, S., Amatriain, X.: Watching television over an ip network. In: Proc. of Usenix/ACM SIGCOMM Internet Measurement Conference (IMC) (October 2008)
6. Cha, M., Rodriguez, P., Moon, S., Crowcroft, J.: On next-generation telco-managed p2p tv architectures. In: Proc. of International Workshop on Peer-To-Peer Systems (IPTPS) (February 2008)
7. Cho, C., Han, I., Jun, Y., Lee, H.: Improvement of channel zapping time in iptv services using the adjacent groups join-leave method. In: Proc. of Int. Conf. on Advanced Communication Technology, ICACT (2004)
8. Haynes, T.W., Hedetniemi, S., Slater, P.: Fundamentals of domination graphs. CRC Press, Boca Raton (1998)
9. He, Y., Shen, G., Xiong, Y., Guan, L.: Optimal prefetching scheme in p2p vod applications with guided seeks. *IEEE Transactions on Multimedia* 11(1), 138–151 (2009)
10. Hei, X., Liang, C., Liang, J., Liu, Y., Ross, K.W.: A measurement study of a large-scale p2p iptv system. *IEEE Transactions on Multimedia* 9(8), 1672–1687 (2007)
11. Jelasity, M., Babaoglu, O.: T-man: Gossip-based overlay topology management. In: ESOA, Intl'l Work. on Engineering Self-Organising Systems (2005)
12. Jelasity, M., Guerraoui, R., Kermarrec, A.-M., van Steen, M.: The peer sampling service: Experimental evaluation of unstructured gossip-based implementations. In: Jacobsen, H.-A. (ed.) *Middleware 2004. LNCS*, vol. 3231, pp. 79–98. Springer, Heidelberg (2004)

13. Lee, J., Lee, G., Seok, S.-H., Chung, B.-D.: Advanced scheme to reduce IPTV channel zapping time. In: Ata, S., Hong, C.S. (eds.) APNOMS 2007. LNCS, vol. 4773, pp. 235–243. Springer, Heidelberg (2007)
14. Li, B., Qu, Y., Keung, Y., Xie, S., Lin, C., Liu, J., Zhang, X.: Inside the new cool-streaming: Principles, measurements and performance implications. In: INFOCOM 2008: Proc. of 27th IEEE Int. Conf. on Computer Communications (April 2008)
15. Qiu, T., Ge, Z., Lee, S., Wang, J., Zhao, Q., Xu, J.: Modeling channel popularity dynamics in a large iptv system. In: Proc. of ACM Sigmetrics (2009)
16. Sentinelli, A., Marfia, G., Gerla, M., Tewari, S., Kleinrock, L.: Will IPTV Ride the Peer-to-Peer Stream? IEEE Communications Magazine 45(6), 86 (2007)
17. Voulgaris, S., Gavridia, D., van Steen, M.: Cyclon: Inexpensive membership management for unstructured p2p overlays. Journal of Network and Systems Management 13(2), 197–217 (2005)
18. Wu, C., Li, B., Zhao, S.: Multi-channel live p2p streaming: Refocusing on servers. In: Proc. of IEEE INFOCOM, pp. 1355–1363 (2008)
19. Wu, D., Liu, Y., Ross, K.W.: Queuing network models for multi-channel p2p live streaming systems. In: Proc. of IEEE INFOCOM (2009)
20. Zheng, C., Shen, G., Li, S.: Distributed prefetching scheme for random seek support in peer-to-peer streaming applications. In: Proc. of the ACM workshop on Advances in peer-to-peer multimedia streaming (2005)

Topic 8

Distributed Systems and Algorithms

Introduction

Dejan Kostić*, Guillaume Pierre*, Flavio Junqueira*, and Peter R. Pietzuch*

In total, fifteen papers were submitted to Topic 8 (including those that were redirected from other Topics), and each paper received four reviews. Five papers were discussed in depth between the chairs. Ultimately, the chairs proposed three papers for acceptance at the PC meeting. The accepted papers tackle a variety of systems and algorithmic problems, with none of the papers tackling some of the “classic” distributed computing topics (e.g., consensus). This was not done as a conscious decision by the chairs; rather, the papers were accepted solely based on quality.

The paper “Active Optimistic Message Logging for Reliable Execution of MPI Applications” focuses on reliability and fault tolerance of MPI applications. An important technique for achieving these goals involves message logging. However, straightforward logging can result in excessive messaging overheads. The authors present a new optimistic message logging protocols, called O2P, that reduces the protocol overhead by logging the dependency information as soon as possible. Evaluations using live experiments with up to 128 processes shows that the technique performs well.

In “A Self-Stabilizing k-Clustering Algorithm Using an Arbitrary Metric,” the authors tackle a problem of grouping a set of nodes in a distributed system based on their locality, or some other metric. This paper proposes a new distributed algorithm for such clustering. The algorithm leverages a Self-Stabilizing Leader Election algorithm, to construct a BFS spanning tree, and color waves for synchronization and inconsistency detection. Authors use simulation to demonstrate the effectiveness of their algorithm.

The paper “Distributed Individual-Based Simulation” describes a new approach to distribute and parallelize individual-based simulations in which entities communicate indirectly with one another through the underlying simulated space. Parallelizing such simulations takes considerable communication overhead since nodes need to exchange information at every iteration. The authors propose communication to be less frequent, e.g., every epoch. This introduces two problems: (1) nodes continue to track particles that have moved to other nodes; (2) particles can become stuck. The authors propose to resolve (1) using shadow cells and (2) using explicit conflict resolution techniques that tackle different scenarios. The experimental results show considerable improvements in execution time when the simulation is parallelized.

Finally, we would like to thank the authors for submitting their work, and the topic chairs and the external reviewers for writing the reviews.

* Topic chairs.

Distributed Individual-Based Simulation

Jiming Liu, Michael B. Dillencourt, Lubomir F. Bic, Daniel Gillen,
and Arthur D. Lander

University of California

Irvine, CA 92697

bic@ics.uci.edu

<http://www.ics.uci.edu/~bic>

Abstract. Individual-based simulations are an important class of applications where a complex system is modeled as a collection of autonomous entities, each having its own identity and behavior in the underlying simulated space. The main drawback of such simulations is that they are extremely compute-intensive. We consider the class of individual-based simulations where the simulated entities interact with one another indirectly through the underlying simulated space, significant performance improvement is attainable through parallelism on a network of machines. We present a data distribution and an approach to reduce the communication overhead, which leads to significant performance improvements while preserving the accuracy of the simulation.

Keywords: individual-based simulation, parallel and distributed computing, performance.

1 Introduction

Individual-based simulations, also known as entity or agent based models, are applications used to simulate the behaviors of a collection of entities, each having its own identity and autonomous behavior, and interacting with one another within a two- or three-dimensional virtual space. At each time step, an entity decides its behavior by interacting with its nearby environment and/or other entities. Typical examples of such applications are interactive battle simulations [1], particle-level simulations in physics [2], traffic modeling [3], and various individual-based simulation models in biology or ecology [4, 5, 6, 7]. Advanced graphics and animation applications, especially those involving large numbers of individuals, such as the animation of a flock of birds, have also taken advantage of spatially-oriented individual-based modeling [8].

There are two types of interaction in individual-based simulation: (1) between entities, and (2) between entities and the environment. Different applications require different types of interaction. For example, n-body problems or the simulation of schooling/flocking behaviors of various animals requires only entity-to-entity interaction because the surrounding space is empty or homogeneous. Ecological simulations, on the other hand typically require both types of

interaction. For example a species of fish interacts with the environment (e.g. by consuming renewable resources) but also with one another. A third type of simulation requires only entity-to-environment interaction. Many particle-level simulations fall into this category because the size of any given particle (e.g. a molecule) is so small relative to the space that collisions can largely be ignored. For example, simulating the propagation of particles (e.g. chemical agents) through living tissue falls into this category. Note however, that while there is no direct entity-to-entity communication, entities still influence each other by modifying the environment, i.e., they communicate with one other indirectly. In particular, multiple entities may compete for a given resource but only one can obtain it. Such simulations where entities communicate with the environment and thus indirectly with one other are the subject of this paper. Specifically, we address the problem of performance: How to distribute and parallelize an individual-based simulation to take advantage of the resources of a computer network or cluster. The major issues that arise in implementing such a parallel computing system include: (1) dividing the problem into small portions, (2) providing a distributed computing environment to support the implementation, and (3) ensuring that the simulation is correct, in the sense that it provides results consistent with what sequential implementation produces.

2 The Particle Diffusion Model

The target application of this research is particle diffusion in a biological intercellular space, based on the molecules diffusion theory of random walks in biology [9]. The biological system consists of cells arranged in a 2-dimensional space. New particles (molecules) enter from one side and, using Brownian motion, propagate through the space between the individual cells. Each cell is equipped with receptors, capable of capturing particles that come close. A captured particle may be released or it may be absorbed after some time. The receptors also move along the cell walls thus aiding the transport of the particles though the space.

Simulated Space. The simulation model represents the cells as equal-sized rectangles arranged in space as shown in Figure 1(a). The left boundary is a closed boundary; particles attempting to go back past the left boundary are bounced back to the simulated space. The right boundary is open; particles that walk across the right boundary disappear from the simulated space. The top and bottom boundaries are connected. Thus, topologically, the space is a cylinder, closed on the left and open on the right.

The Cells. Each cell (Figure 1(b)) is a square with a size of $10\mu\text{m}$ by $10\mu\text{m}$ and the distance between cells is one tenth of the cell size, i.e., $1\mu\text{m}$. The wall of each cell is divided into a number of segments (generally 20).

The Receptors. Receptors reside on cell walls. Each receptor has two states: free or occupied. When a receptor captures a particle its state changes from free

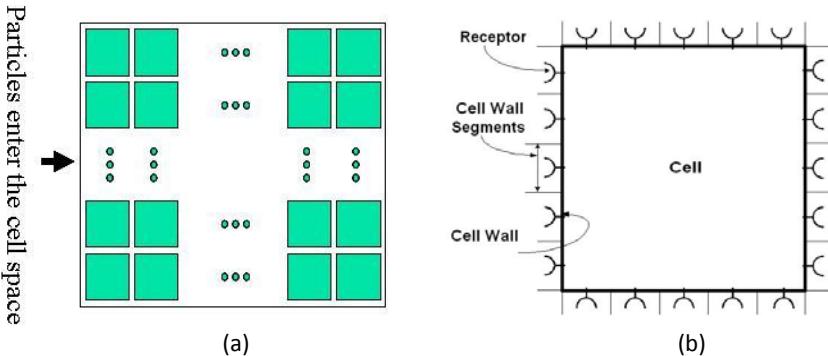


Fig. 1. (a) The simulated space; (b) One cell

to occupied. When the stuck particle is released or degraded (absorbed), the receptor changes its state back to free. Receptors, both free and occupied, move between neighboring cell segments at a predefined rate at every simulation time step. Thus the number of receptors in a given segment varies over time. This movement balances the number of free and occupied receptors on the cell segments.

The Particles. Particles are identified in the system by their position and state. The position of a particle is defined by its coordinates, x and y , in the simulated space. A particle does not own a territory. Therefore the density of particles in the simulated space has no limit. The total number of particles entering into the system (as well as many other spatial or behavioral characteristics) can be varied to simulate different cases, according to biological assumptions.

New particles enter into the simulated space periodically during the course of simulation. A particle starts as a free particle and can travel freely through the alleys (intercellular spaces) or be captured by a free receptor on a cell segment.

In the basic model (simulating the Brownian motion), each free particle moves during every iteration for a fixed distance (the micro step, or $10^{-9}m$) in one of four compass directions chosen randomly with equal probability. If it comes within a certain distance of a free receptor, it is captured. The main problem with this model is that it is too slow. For example, to simulate one biological minute requires on the order of 10^8 iterations. An improved model has been presented in [10]. This moves particles in macro steps using a Gaussian distribution and it determines the events of capture, release, and degradation probabilistically. This model improves performance by several orders of magnitude. The following sections describe how performance can be improved further by distributing the data and parallelizing the code.

3 Basic Distributed Implementation

The simulated space is partitioned into horizontal strips (using an Eulerian decomposition) and each strip is assigned to a different node. This partitioning preserves the locality of particles, which minimizes communication. It also preserves

```

1. while(current time < simulation time)  {
2.   Receive messages from left and right neighbor
3.   Update own simulation space
4.   Move Receptors for all cells
5.   Process Stuck Particles
6.   Process all Free Particles
7.   Move all emigrating particles into left and right messages
8.   Send messages to left and right neighbor
9.   Increment current time
10. }

```

Fig. 2. Basic simulation cycle

load balance, because there are large numbers of particles, all of which have the same behavior, and thus every node manages the same number of particles on average. Each node executes the same code shown in Figure 2 which implements a basic time-driven simulation. At the beginning of each iteration of the simulation run, each node waits for a message from its left and right neighbors (line 2). These messages contain particles that have migrated into the space managed by the current node from its neighbors. The node places them into its simulation space (line 3). Next the node processes all receptors by moving them probabilistically along the cell walls according to the simulation parameters (line 4). Some of the stuck particles are degraded and disappear from the system while other are released as free particles (line 5). Next all free particles are processed by moving them to their respective new positions and determining if they become stuck (line 6). All particles whose new position is outside of the nodes assigned space are extracted (line 7) and sent to the corresponding neighboring nodes (line 8). Finally, the simulation time is incremented to the next step (line 9).

4 Reducing Communication Overhead

Our goal is to distribute and parallelize the simulation model to speed up its execution. The problem with the straight-forward implementation outlined in the previous section is its excessive communication overhead: Nodes must exchange information at every iteration. Because of this overhead, this implementation is actually slower than the sequential implementation. The obvious solution is to exchange information less frequently. We define an epoch as a time interval (number of iterations) between two consecutive data exchanges. This optimistic version of the simulation reduces communication overhead but introduces two new problems:

1. A particle can move across to a neighbor node during any iteration within an epoch but because nodes do not exchange information at every iteration, the original node must continue tracking the particle even though it is no longer in its assigned region. That problem can be solved by introducing shadow cells.

- Because a node does not find out about incoming particles until the end of each epoch when information is exchanged, it is operating with out-of-date information, which can cause particles to become stuck when they should not, or vice versa. This must be resolved using an explicit conflict resolution scheme at every epoch.

4.1 Shadow Cells

We create shadow cells to extend the local node boundary in order to allow the node to track particles that have emigrated from its assigned space. A shadow cell is simply a copy of the neighbor cell taken at the beginning of the epoch. Consider for example a rectangular space of 5x5 cells. Assume the each row is mapped on a different node. That is, node 1 holds the cells 11, 12, 13, 14, 15; node 2 holds the cells 21, 22, 23, 24, 25; node 3 holds the cells 31, 32, 33, 34, 35; and so on. Then node 2, in addition to its own cells, will also hold copies of the cells 11, ..., 15 (of node 1) and 31, ..., 35 (of node 3) as shadow cells. Note that nodes 1 and 5 are also neighbors because the space is an open cylinder. Thus node 1 will hold copies of the cells 51, ..., 55 (of node 5) and 21, ..., 25 (of node 2). Shadow cells are updated at every epoch when nodes exchange information with one another.

4.2 Conflict Scenarios

The shadow cells allow nodes to continue tracking particles that have left their assigned space. The problem is that the receiving nodes will not know about the new incoming particles until the end of the epoch. As a result, they may take actions that would be different from those taken if the new particles were visible immediately. Such conflict situations must be recognized and corrected at the end of every epoch. There are two different scenarios under which a conflict occurs.

Scenario 1: A free particle is captured when it should not Figure 3 shows a situation where, due to out-of-date information, a particle is captured by a receptor that is no longer free. The left half of the diagram shows the sequential execution. At T2, a free receptor captures a particle (solid circle). At T3, another particle (hollow circle) approaches the same receptor but continues as free.

The right half of the diagram shows a distributed execution of the same situation. Node 1 controls the top row of the original space; node 2 controls the bottom row. The corresponding shadow cells are labeled with the letter S. At time T0, the beginning of the epoch, the system is synchronized by exchanging data between neighbor nodes. Thus the shadow cells are exact copies of their real counterparts. At time T1, one particle (hollow circle) walks across the space boundary of Node 2. It enters the shadow region of Node 1 but that node does not see it in its real space. At time T2, Node 1 captures the solid particle but this is not visible to Node 2. Consequently, at time T3, Node 2 captures the

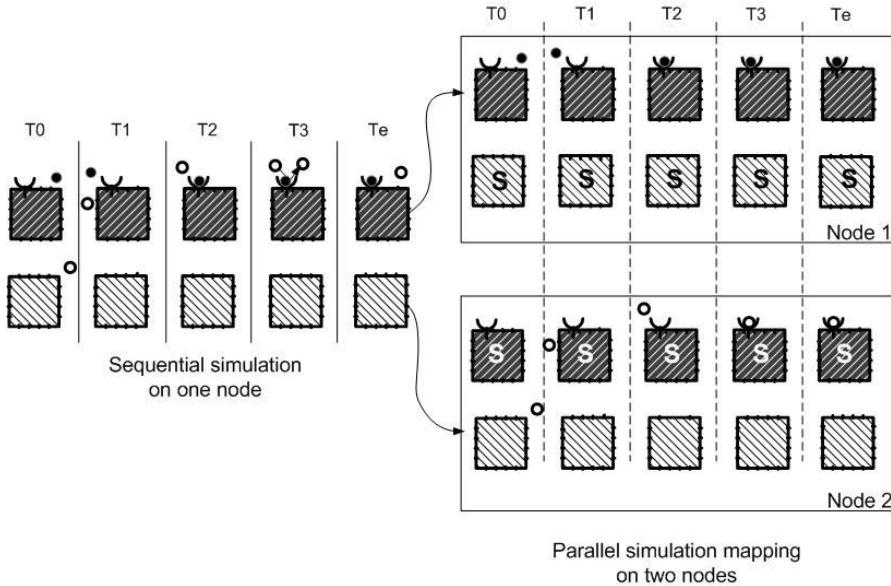


Fig. 3. Sequential and parallel execution of scenario 1

hollow particle. The resulting state at the end of the epoch, T_e , is inconsistent with the sequential execution because both particles have been captured by the same receptor. The conflict must explicitly be resolved.

Scenario 2: A free particle is not captured when it should Figure 4 shows a situation where, due to out-of-date information, a node fails to capture a particle by a free receptor. The left half of the diagram shows the sequential execution. At T_1 , a previously captured particle (solid circle) is degraded and the receptor becomes free. As a result, it captures a new particle (hollow circle) that approaches it at T_2 .

In the distributed implementation, Node 2 does not learn about the degradation of the solid particle; the receptor appears as occupied in the shadow cell for the duration of the epoch. Consequently, Node 2 fails to capture the hollow particle approaching the receptor at T_3 . The resulting situation at the end of the epoch, T_e , is thus inconsistent with the sequential execution.

5 Conflict Resolution

5.1 Solution to Scenario 1

The conflict resolution scheme is to take snapshots of every event of particle capture that occurs during the epoch; these remain tentative until the end of the epoch, when the snapshots are exchanged and each capture either confirmed or rejected as part of the conflict resolution.

The record of every capture contains information about the particles location prior to capture, including the cell segment and the number of free receptors of the capturing segment, and other information necessary to replay the event at the time of conflict resolution. The record also differentiates between two different types of capture:

Own: The particle was captured within the cell belonging to the node

Shadow: The particle was captured within a shadow cell

At the end of every epoch, all records of tentatively captured particles are exchanged among neighboring nodes. A conflict occurs whenever one node records a capture of type own while its neighbor records a capture of type shadow in the neighbors shadow copy. This indicates that both nodes performed a capture of some particle in the same cell but, because their decisions were based on out-of-date information, one of the captures may not be valid.

Whenever a conflict is detected, each node reprocesses all tentatively stuck particles based on the now accurate information contained in the exchanged records. The goal of the resolution is to find the tentatively stuck particles that should not become stuck, and then release them back to remain as free particles.

To illustrate the above conflict resolution process, consider again the scenario of Figure 3. Node 1 records a capture at time T2. This capture is of type own because the particle resides in the cell assigned to Node 1. Node 2 records a capture at time T3; this is of type shadow because the particle resides in the shadow cell corresponding to the actual cell of Node 1. The two records are exchanged by the two nodes at the end of the epoch and, because they are of different types, conflict resolution is carried out by both nodes:

1. At time T2, the first record is processed. The information in the tag matches the actual local information, i.e., there is a free receptor available in that cell segment at this time. Consequently, the tentatively stuck particle is confirmed as stuck and the free receptor becomes occupied.
2. At time T3 the second record is processed. At this time, there is no free receptor remaining in the cell segment (as a result of the previous step) and hence the second tentatively stuck particle cannot become stuck. It is released as a free particle and its new location at the end of the epoch is calculated.

The result of the conflict resolution then matches that of the sequential simulation shown in Figure 5, where only the first particle has been captured.

5.2 Solution to Scenario 2

Scenario 2 is more difficult because it is caused by the absence of a capture event. Obviously, there is no record of such a non-occurring event (it is not possible to record all events that have not occurred.) Hence there is no information for the conflict resolution to use. Our solution to this problem is to overstate the number of free receptors in the shadow cells at the beginning of the epoch. Overstating this number by one allows one additional particle to be captured that should

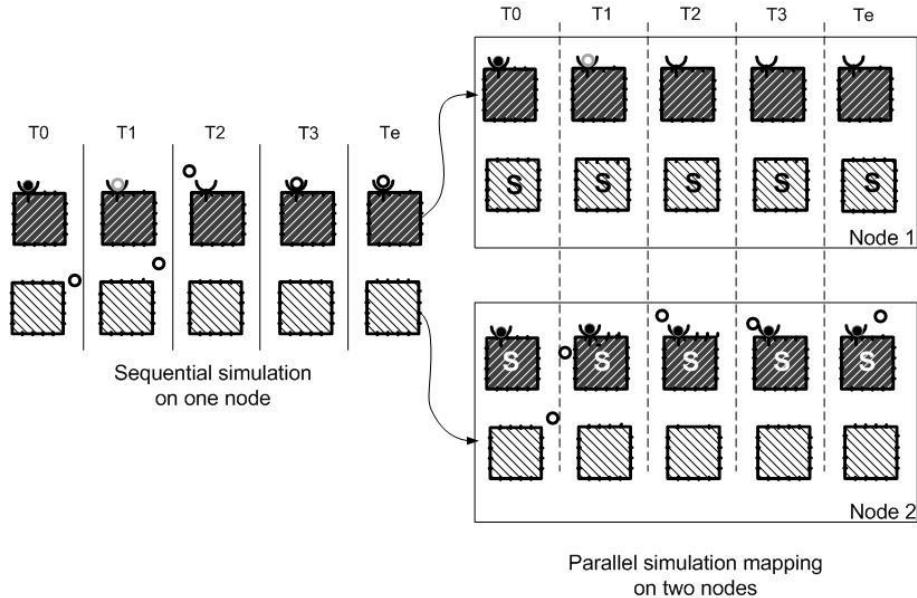


Fig. 4. Sequential and parallel execution of scenario 2

not have been captured. If this occurs, however, the conflict resolution scheme described previously will detect and correct this error.

To illustrate this conflict resolution process, consider again the scenario of Figure 4. The number of free receptors in the shadow cell maintained by Node 2 is now artificially increased to 2 instead of 1. At time T1, Node 1 records the degradation of the captured particle; this event is of type own. At time T3, Node 2 records a capture of type shadow. This is now possible because there are 2 receptors, one of which is still free.

The two records are exchanged by the two nodes at the end of the epoch and, because they are of different types, conflict resolution is carried out by both nodes. The reprocessing now uses the actual (not overstated) numbers of free receptors: At time T1, the particle is degraded and the receptor becomes free. At time T3, the second capture is also correctly confirmed because of the availability of the receptor released during the previous step. The result of the conflict resolution then matches that of the sequential simulation shown in Figure 4, where both particles have been sequentially captured.

6 Performance Evaluation

The main objective of the distributed implementation is to increase performance and the most important measure is the speedup achieved over the corresponding sequential implementation. This is achieved by increasing the epoch length, which reduces communication overhead. However, with increased epoch length

the probability of conflict increases. Furthermore, some of the conflicts become unrecoverable. Specifically, a node cannot track a particle beyond its shadow cells; when a particle walks too far during the epoch, it becomes lost. Similarly, overstating the number of receptors in the shadow cells by 1 (earlier scenario 2) can handle only a single conflict. Overstating by more than one would handle multiple conflicts but would also result in additional recovery overhead caused by the phantom capture events caused by the additional (non-existent) receptors. For all these reasons, the epoch should be kept as short as possible. Hence the ultimate goal is find the ideal epoch length where the combined effects of communication and conflict resolution are minimized.

Experiment 1. The simulated space in this experiment consists of 50 cells organized in 5 rows with 10 cells in each row. New particles enter into the system from the left boundary of the simulated space at a rate of 0.001 particles per iteration per cell row. The experiment runs for 1,000,000 iterations thus simulating a total of 5000 particles. The parallel implementation uses 5 nodes with each cell row assigned to a different node. We run the experiment with the epoch lengths of 10, 100, 200, 500, 1000, 2000, 3000 and 5000.

Figure 5(a) shows the execution times for the sequential and parallel simulations. Figure 5(b) shows the corresponding speedups. The execution time for the sequential simulation is 1303 seconds. The break-even point where the sequential and the parallel implementation need the same time is with an epoch length of less than 10. There is a steep improvement in performance when the epoch length is increased from 10 toward 1000. Past that point, the improvement is only marginal because both communication and conflict resolution overheads are very small. The speedup with epoch lengths of 5000 is 2.62 (on 5 nodes).

Experiment 2. In this experiment, we increase the simulated space to 100 cells organized in 10 rows with 10 cells in each row. Each row is mapped on a separate node. New particles enter into the system at a rate of 0.001 per iteration per cell row. There are total of 10,000 particles simulated in this experiment. The epoch lengths are the same as in the previous experiments.

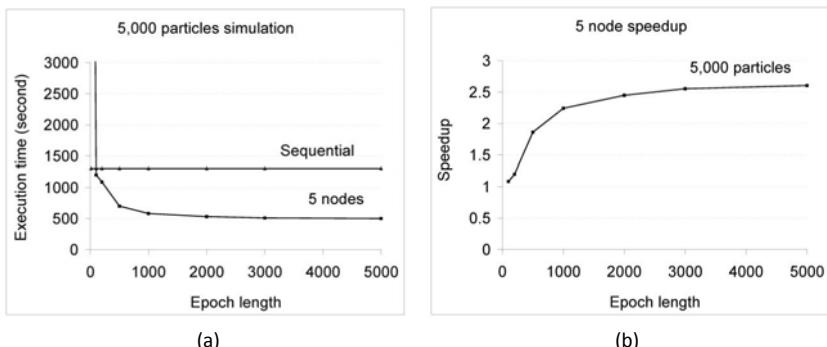


Fig. 5. Execution times and speedup of experiment 1 on 5 nodes

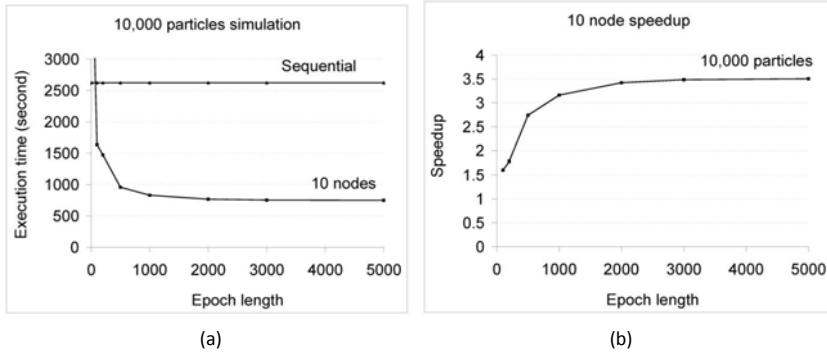


Fig. 6. Execution times and speedup of experiment 2 on 10 nodes

Figure 6 shows the execution times and speedups of this experiment. The execution time of the sequential simulation is 2625 seconds with a break-even point between 10 and 100 iterations per epoch. As before, the speedup improves significantly when epoch length increases toward 1000 but then becomes flat. The maximum speedup attained is 3.5. This is significantly better than the 2.62 speedup with 5 nodes (experiment 1) but 10 nodes were required to achieve this.

Experiment 3. This experiment investigates if a better speedup is obtainable when the problem size (in terms of the number of particles in the simulation) is greater. We keep the same simulated space but increase the number of incoming particles to 0.004 per iteration per row. There are now 40,000 particles simulated in this experiment, 4 times more than it in experiment 2.

Figure 7 shows the execution times and the speedups for this experiment. The execution time of the sequential simulation is 20712 seconds and the break-even point is again between 10 and 100 iterations. For comparison, Figure 7(b) also includes the speedup of experiment 2 (10,000 particles). The maximum attained

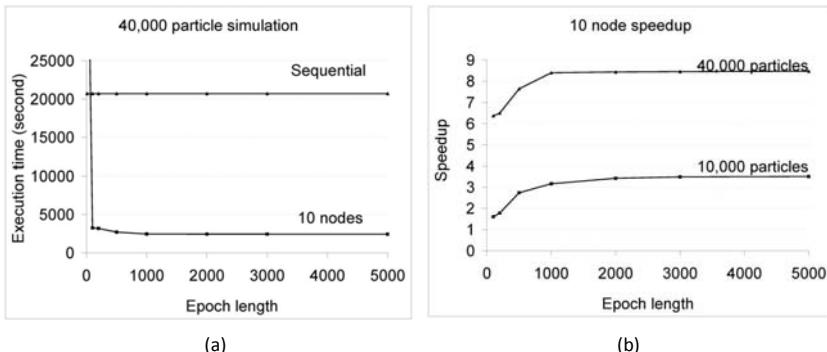


Fig. 7. Execution times and speedup of experiment 3 on 10 nodes

Table 1. Comparison of the speedup and accuracy at different epoch lengths

Iterations	500	1000	2000	3000	5000
Accuracy	100%	100%	99.7%	94.8%	92.1%
5 node speedup	4.06	4.43	4.43	4.41	4.43
10 node speedup	7.64	8.40	8.44	8.45	8.47

speedup of experiment 3 is 8.47 (on 10 nodes), which is significantly better than the speedup of 3.5 (attained on 5 nodes in experiment 2).

Performance versus Accuracy. These experiments show that the simulation is highly scalable. When nodes are added or the number of simulated particles is increased, the simulation attains significant speedup, especially when the epoch length increases past 1000 iterations. Increasing the epoch length past 1000 still gains some performance but lowers the accuracy. That is, some portion of the simulated particles are in different states or positions in the sequential and distributed implementations. This is because the conflict resolution scheme cannot resolve certain situations that occur when the epoch is very long.

Table 1 shows the trade-offs between performance and accuracy. With an epoch length of 1000 or less, accuracy is still at 100%. That is, the number and locations of all particles are identical in the sequential and distributed implementations. Between 1000 and 2000, some unrecoverable discrepancies begin to appear. On the other hand, the speedup increases dramatically when the epoch length is increase from 1 toward 1000 but is only marginal past 1000. This gives the user the option of choosing between 100% accuracy with a slightly lower performance or to aim for maximum speedup with a slight loss of accuracy.

7 Conclusions

Individual-based simulations are an important class of applications that are able to general result unattainable from statistics-based simulation models and hence are used frequently in various science disciplines. The main drawback of such simulations is that they are extremely compute-intensive. This paper considered a class of individual-based simulations where the simulated particles interact with one another indirectly by interacting with the underlying simulated space (biological cells and their receptors, in our case). We have demonstrated that an Eulerian space partitioning allows the simulation to effectively utilize multiple nodes, provided the communication overhead is decreased by the introduction of epochs and the necessary conflict resolution mechanisms to correct any inaccuracy resulting from the delayed information exchange.

The implementation presented in this paper is an optimistic simulation [11] but it differs from other approaches in several important ways. First, our simulation is time-driven rather than discrete-even-driven. All processes proceed at the same pace in virtual time and thus one cannot get ahead of another. Second, the periodic exchange of information is delayed by the epoch length,

which means that, by design, all messages are straggler messages and require corrective action. However, a complete rollback is not necessary. Instead, we are able to correct individual conflicts at specific receptors through the mechanisms introduced in this paper. Thus the cost of the optimistic view and the need for recovery (conflict resolution) is by far outweighed by the gains in performance resulting from the dramatically reduced communication overhead.

References

1. DIS Steering Committee, The DIS vision: A map to the future of distributed simulation. Institute for Simulation and Training (1994)
2. Hockney, R.W., Eastwood, J.W.: Computer Simulations using Particles. IOP Publishing Ltd., Bristol (1988)
3. Resnick, M.: Changing the centralized mind. *Technology Review*, 33–40 (1994)
4. Huston, M., DeAngelis, D., Post, W.: New computer models unify ecological theory. *BioScience* 38(10), 682–691 (2001)
5. Huth, A., Wissel, C.: The simulation of the movement of fish schools. *Journal of Theoretical Biology* 156, 365–385 (1992)
6. Villa, F.: New computer architectures as tools for ecological thought. *Trends in Ecology and Evolution (TREE)* 7(6), 179–183 (1992)
7. Hartvigsen, G., Levin, S.A.: Evolution and spatial structure interact to influence plant-herbivore population and community dynamics. *Proc. Roy. Soc. London Ser. B* 264, 1677–1685 (1997)
8. Reynolds, C.W.: Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics* 21(4), 25–34 (1987)
9. Berg, H.C.: Random Walks in biology. Princeton University Press, Princeton (1993)
10. Liu, J.: Distributed Individual-Based Simulation, PhD Dissertation, Dept. of Computer Sci, University of California, Irvine (2009)
11. Fujimoto, R.: Parallel and Distributed Simulation Systems. In: Proc. 2001 Winter Simulation Conference, vol. 2 (2001)

A Self-stabilizing K-Clustering Algorithm Using an Arbitrary Metric

Eddy Caron^{1,2}, Ajoy K. Datta³, Benjamin Depardon^{1,2},
and Lawrence L. Larmore³

¹ University of Lyon. LIP Laboratory. UMR CNRS - ENS Lyon

² - INRIA - UCBL 5668, France

³ University of Nevada Las Vegas, USA

Abstract. Mobile *ad hoc* networks as well as grid platforms are distributed, changing and error prone environments. Communication costs within such infrastructures can be improved, or at least bounded, by using *k-clustering*. A *k*-clustering of a graph, is a partition of the nodes into disjoint sets, called clusters, in which every node is distance at most *k* from a designated node in its cluster, called the *clusterhead*. A self-stabilizing asynchronous distributed algorithm is given for constructing a *k*-clustering of a connected network of processes with unique IDs and weighted edges. The algorithm is comparison-based, takes $O(nk)$ time, and uses $O(\log n + \log k)$ space per process, where *n* is the size of the network. To the best of our knowledge, this is the first distributed solution to the *k*-clustering problem on weighted graphs.

Keywords: K-clustering, self-stabilization, weighted graph.

1 Introduction

Nowadays distributed systems are built over a large number of resources. Overlay structures require taking into account locality among the entities they manage. For example, communication time between resources is the main performance metric in many systems. A cluster structure facilitates the spatial *reuse of resources* to increase system capacity. Clustering also helps routing and can improve the efficiency of a parallel software if it runs on a cluster of well connected resources. Another advantage of clustering is that many changes in the network can be made locally, *i.e.*, restricted to particular clusters.

Many applications require that entities are grouped into clusters according to a certain distance which measures proximity with respect to some relevant criterion; the clustering will result in clusters with similar or bounded readings. We are interested in two particular fields of research which can make use of resource clustering: mobile *ad hoc* networks (MANET) and application deployment on grid environments.

In MANET, scalability of large networks is a critical issue. Clustering can be used to design a low-hop backbone network in MANET with routing facilities provided by clustering. However, using only hops, *i.e.*, the number of links in the path between two processes, may hide the true communication time between two nodes.

A major aspect of grid computing is the deployment of grid middleware. The hop distance is used as a metric in some applications, but it may not be relevant in many platforms, such as a grid. Using an arbitrary metric (*i.e.*, a weighted metric) is a reasonable option in such heterogeneous distributed systems. Distributed grid middleware, like DIET [1] and GridSolve [2] can make use of accurate distance measurements to do efficient job scheduling.

Another important aspect is that both MANET and grid environments are highly dynamic systems: nodes can join and leave the platform anytime, and may be subject to errors. Thus, designing an efficient fault-tolerant algorithm which clusters the nodes according to a given distance k , and which can dynamically adapt to any change, is a necessity for many applications, including MANET and grid platforms.

Self-stabilization [3] is a desirable property of fault-tolerant systems. A self-stabilizing system, regardless of the initial states of the processes and initial messages in the links, is guaranteed to converge to the intended behavior in finite time. Self-stabilization has been shown to be a powerful tool to design dynamic systems. As MANET and grid platforms are dynamic and error prone infrastructures, self-stabilization is a good approach to design efficient algorithms.

1.1 The k -Clustering Problem

We now formally define the problem solved in this paper. Let $G = (V, E)$ a connected graph (network) consisting of n nodes (processes), with positively weighted edges. For any $x, y \in V$, let $w(x, y)$ be the *distance* from x to y , defined to be the least weight of any path from x to y . We will assume that the edge weights are integers. We also define the radius of a graph G as follows: $\text{radius}(G) = \min_{x \in V} \max_{y \in V} \{w(x, y)\}$.

Given a non-negative integer k , we define a k -cluster of G to be a non-empty connected subgraph of G of radius at most k . If C is a k -cluster of G , we say that $x \in C$ is a *clusterhead* of C if, for any $y \in C$, there is a path of length at most k in C from x to y .

We define a k -clustering of G to be a partitioning of V into k -clusters. The k -clustering problem is then the problem of finding a k -clustering of a given graph.¹ In this paper, we require that a k -clustering specifies one node, which we call the *clusterhead* within each cluster, which is within k of all nodes of the cluster, and a *shortest path tree* rooted at the clusterhead which spans all the nodes of the cluster.

¹ There are several alternative definitions of k -clustering, or the k -clustering problem, in the literature.

A set of nodes $D \subseteq V$ is a *k-dominating set*² of G if, for every $x \in V$, there exists $y \in D$ such that $w(x, y) \leq k$. A *k*-dominating set determines a *k*-clustering in a simple way; for each $x \in V$, let $Clusterhead(x) \in D$ be the member of D that is closest to x . Ties can be broken by any method, such as by using IDs. For each $y \in D$, $C_y = \{x : Clusterhead(x) = y\}$ is a *k*-cluster, and $\{C_y\}_{y \in D}$ is a *k*-clustering of G . We say that a *k*-dominating set D is *optimal* if no *k*-dominating set of G has fewer elements than D . The problem of finding an optimal *k*-dominating set is known to be \mathcal{NP} -hard [5].

1.2 Related Work

To the best of our knowledge, there exist only three asynchronous distributed solutions to the *k*-clustering problem in mobile *ad hoc* networks, in the comparison based model, *i.e.*, where the only operation allowed on IDs is comparison. Amis *et al.* [5] give the first distributed solution to this problem. The time and space complexities of their solution are $O(k)$ and $O(k \log n)$, respectively. Spohn and Garcia-Luna-Aceves [6] give a distributed solution to a more generalized version of the *k*-clustering problem. In this version, a parameter m is given, and each process must be a member of m different *k*-clusters. The *k*-clustering problem discussed in this paper is then the case $m = 1$. The time and space complexities of the distributed algorithm in [6] are not given. Fernandess and Malkhi [7] give an algorithm for the *k*-clustering problem that uses $O(\log n)$ memory per process, takes $O(n)$ steps, provided a BFS tree for the network is already given. The first self-stabilizing solution to the *k*-clustering problem was given in [8]; it takes $O(k)$ time and $O(k \log n)$ space. However, this algorithm only deal with the hop metric, and is thus unable to deal with more general weighted graphs.

1.3 Contributions and Outline

Our solution, Algorithm Weighted-Clustering, given in Sect. 3, is partially inspired by that of Amis *et al.* [5], who use simply the hop distance instead of arbitrary edge weights. Weighted-Clustering uses $O(\log n + \log k)$ bits per process. It finds a *k*-dominating set in a network of processes, assuming that each process has a unique ID, and that each edge has a positive weight. It is also self-stabilizing and converges in $O(nk)$ rounds. When Algorithm Weighted-Clustering stabilizes, the network is divided into a set of *k*-clusters, and inside each cluster, the processes form a shortest path tree rooted at the clusterhead.

In Sect. 2, we describe the model of computation used in the paper, and give some additional needed definitions. In Sect. 3, we define the algorithm Weighted-Clustering, and give its time and space complexity. We also show an example execution of Weighted-Clustering in Sect. 4. Finally, we present some simulation results in Sect. 5 and conclude the paper in Sect. 6.

² Note that this definition of the *k*-dominating set is different than another well known problem consisting in finding a subset $V' \subseteq V$ such that $|V'| \leq k$, and such that $\forall v \in V - V', \exists y \in V' : (x, y) \in E$. [4].

2 Model and Self-stabilization

We are given a connected undirected network of size $n \geq 2$, and a distributed algorithm \mathcal{A} on that network. Each process P has a unique ID, $P.id$, which we assume can be written with $O(\log n)$ bits.

The *state* of a process is defined by the values of its registers. A *configuration* of the network is a function from processes to states; if γ is the current configuration, then $\gamma(P)$ is the current state of each process P . An *execution* of \mathcal{A} is a sequence of states $e = \gamma_0 \mapsto \gamma_1 \mapsto \dots \mapsto \gamma_i \dots$, where $\gamma_i \mapsto \gamma_{i+1}$ means that it is possible for the network to change from configuration γ_i to configuration γ_{i+1} in one step. We say that an execution is *maximal* if it is infinite, or if it ends at a *sink*, *i.e.*, a configuration from which no execution is possible.

The *program* of each process consists of a set of registers and a finite set of actions of the following form: $< \text{label} > :: < \text{guard} > \longrightarrow < \text{statement} >$. The *guard* of an action in the program of a process P is a Boolean expression involving the variables of P and its neighbors. The *statement* of an action of P updates one or more variables of P . An action can be executed only if it is *enabled*, *i.e.*, its guard evaluates to true. A process is said to be *enabled* if at least one of its actions is enabled. A step $\gamma_i \mapsto \gamma_{i+1}$ consists of one or more *enabled* processes executing an action.

We use the *shared memory/composite atomicity model* of computation [39]. Each process can read its own registers and those of its neighbors, but can write only to its own registers; the evaluations of the guard and executions of the statement of any action is presumed to take place in one atomic step.

We assume that each transition from a configuration to another is driven by a *scheduler*, also called a *daemon*. At a given step, if one or more processes are enabled, the daemon selects an arbitrary non-empty set of enabled processes to execute an action. The daemon is thus *unfair*: even if a process P is continuously enabled, P might never be selected by the daemon, unless, at some step, P is the only enabled process.

We say that a process P is *neutralized* during a step, if P is enabled before the step but not after the step, and does not execute any action during that step. This situation could occur if some neighbors of P change some of their registers in such a way as to cause the guards of all actions of P to become false.

We use the notion of *round* [10], which captures the speed of the slowest process in an execution. We say that a finite execution $\varrho = \gamma_i \mapsto \gamma_{i+1} \mapsto \dots \mapsto \gamma_j$ is a *round* if the following two conditions hold: (i) Every process P that is enabled at γ_i either executes or becomes neutralized during some step of ϱ , (ii) The execution $\gamma_i \mapsto \dots \mapsto \gamma_{j-1}$ does not satisfy condition (i). We define the *round complexity* of an execution to be the number of disjoint rounds in the execution, possibly plus one more if there are some steps left over.

The concept of *self-stabilization* was introduced by Dijkstra [3]. Informally, we say that \mathcal{A} is *self-stabilizing* if, starting from a completely arbitrary configuration, the network will eventually reach a legitimate configuration.

More formally, we assume that we are given a *legitimacy predicate* $\mathcal{L}_{\mathcal{A}}$ on configurations. Let $\mathbb{L}_{\mathcal{A}}$ be the set of all *legitimate* configurations, *i.e.*, configurations

which satisfy $\mathcal{L}_{\mathcal{A}}$. Then we define \mathcal{A} to be *self-stabilizing* to $\mathbb{L}_{\mathcal{A}}$, or simply *self-stabilizing* if $\mathbb{L}_{\mathcal{A}}$ is understood, if the following two conditions hold: (i) (Convergence) Every maximal execution contains some member of $\mathbb{L}_{\mathcal{A}}$, (ii) (Closure) If an execution e begins at a member of $\mathbb{L}_{\mathcal{A}}$, then all configurations of e are members of $\mathbb{L}_{\mathcal{A}}$. We say that \mathcal{A} is *silent* if every execution is finite. In other words, starting from an arbitrary configuration, the network will eventually reach a *sink*, *i.e.*, a configuration where no process is enabled.

3 The Algorithm Weighted-Clustering

In this section, we present Weighted-Clustering, a self-stabilizing algorithm that computes a k -clustering of a weighted network of size n .

3.1 Overview of Weighted-Clustering

A process P is chosen to be a *clusterhead* if and only if, for some process Q , P has the smallest ID of any process within a distance k of Q . The set of clusterheads so chosen is a k -dominating set, and a clustering of the network is then obtained by every process joining a shortest path tree rooted at the nearest clusterhead. The nodes of each such tree form one k -cluster.

Throughout, we write \mathcal{N}_P for the set of all neighbors of P , and $\mathcal{U}_P = \mathcal{N}_P \cup \{P\}$ the closed neighborhood of P . For each process P , we define the following values:

$$\text{MinHop}(P) = \min \{ \min \{ w(P, Q) : Q \in \mathcal{N}_P \}, k + 1 \}$$

$$\text{MinId}(P, d) = \min \{ Q.id : w(P, Q) \leq d \}$$

$$\text{MaxMinId}(P, d) = \max \{ \text{MinId}(Q, k) : w(P, Q) \leq d \}$$

$$\text{Clusterhead_Set} = \{P : \text{MaxMinId}(P, k) = P.id\}$$

$$\text{Dist}(P) = \min \{ w(P, Q) : Q \in \text{Clusterhead_Set} \}$$

$$\text{Parent}(P) = \begin{cases} P.id & \text{if } P \in \text{Clusterhead_Set} \\ \min \left\{ Q.id : \begin{array}{l} (Q \in \mathcal{N}_P) \wedge \\ (Dist(Q) + w(P, Q) = Dist(P)) \end{array} \right\} & \text{otherwise} \end{cases}$$

$$\text{Clusterhead}(P) = \begin{cases} P.id & \text{if } P \in \text{Clusterhead_Set} \\ \text{Clusterhead}(\text{Parent}(P)) & \text{otherwise} \end{cases}$$

The *output* of Weighted-Clustering consists of shared variables $P.parent$ and $P.clusterhead$ for each process P . The output is *correct* if $P.parent = \text{Parent}(P)$ and $P.clusterhead = \text{Clusterhead}(P)$ for each P . Hence, the previous values define the sequential version of our algorithm. Weighted-Clustering is self-stabilizing. Although it can compute incorrect output, the output shared variables will eventually stabilize to their correct values.

3.2 Structure of Weighted-Clustering: Combining Algorithms

The formal definition of Weighted-Clustering requires 26 functions and 15 actions, and thus it is difficult to grasp the intuitive principles that guide it. In this conference paper, we present a broad and intuitive explanation of how the

algorithm works. Technical details of Weighted-Clustering can be found in our research report [11] which also contains proofs of its correctness and complexity.

Weighted-Clustering consists of the following four phases.

Phase 1, Self-Stabilizing Leader Election (SSLE). We make use of an algorithm SSLE, defined in [12], which constructs a breadth-first-search (BFS) spanning tree rooted at the process of lowest ID, which we call *Root_BFS*. The BFS tree is defined by pointers $P.parent_BFS$ for all P , and is used to synchronize the second and third phases of Weighted-Clustering. SSLE is self-stabilizing and silent. We do not give its details here, but instead refer the reader to [12].

The BFS tree created by SSLE is used to implement an efficient broadcast and convergecast mechanism, which we call *color waves*, used in the other phases.

Phase 2 and 3, A non-silent self-stabilizing algorithm Interval. Given a positively weighted connected network with a rooted spanning tree, a number $k > 0$, and a function f on processes, Interval computes $\min \{f(Q) : w(P, Q) \leq k\}$ for each process P in the network, where $w(P, Q)$ is the minimum weight of any path through the network from P to Q .

- **Phase 2, MinId**, which computes, for each process P , $MinId(P, k)$, the smallest ID of any process which is within distance k of P . The color waves, *i.e.*, the Broadcast-convergecast waves on the BFS tree computed by SSLE, are used to ensure that (after perhaps one unclean start) MinId begins from a clean state, and also to detect its termination. MinId is not silent; after computing all $MinId(P, k)$, it resets and starts over.
- **Phase 3, MaxMinId**, which computes, using Interval, for each process P , $MaxMinId(P, k)$, the largest value of $MinId(Q, k)$ of any process Q which is within distance k of P .

The color waves are timed so that the computations of MinId and MaxMinId alternate. MinId will produce the correct values of $MinId(P, k)$ during its first complete execution after SSLE finishes, and MaxMinId will produce the correct values of $MaxMinId(P, k)$ during its first complete execution after that.

Phase 4, Clustering. A silent self-stabilizing algorithm which computes the clusters given *Clusterhead_Set*, which is the set of processes P for which $MaxMinId(P, k) = P.id$. *Clustering* runs concurrently with MinId and MaxMinId, but until those have both finished their first correct computations, *Clustering* may produce incorrect values. *Clusterhead_Set* eventually stabilizes (despite the fact that MinId and MaxMinId continue running forever), after which Clustering has computed the correct values of $P.clusterhead$ and $P.parent$ for each P .

3.3 The BFS Spanning Tree Module SSLE

It is only necessary to know certain conditions that will hold when SSLE converges.

- There is one *root* process, which we call *Root_BFS*, which SSLE chooses to be the process of smallest ID in the network.
- $P.dist_BFS$ = the length (number of hops) of the shortest path from P to *Root_BFS*.

$$- P.parent_BFS = \begin{cases} P.id & \text{if } P = Root_BFS \\ \min \left\{ Q.id : \begin{array}{l} (Q \in \mathcal{N}_P) \wedge \\ (Q.dist_BFS + 1) = P.dist_BFS \end{array} \right\} & \text{otherwise} \end{cases}$$

SSLE converges in $O(n)$ rounds from an arbitrary configuration, and remains silent, thus throughout the remainder of the execution of Weighted-Clustering, the BFS tree will not change.

3.4 Error Detection and Correction

There are four colors, 0, 1, 2, and 3. The BFS tree supports the color waves; these scan the tree up and down, starting from the bottom of the tree for colors 0 and 2, and from the top for 1 and 3. The color waves have two purposes: they help the detection of inconsistencies within the variables, and allow synchronization of the different phases of the algorithm. Before the computation of clusterheads can be guaranteed correct, all possible errors in the processes must be corrected. Three kinds of errors are detected and corrected:

- Color errors: $P.color$ is 1 or 3, and the color of its parent in the BFS tree is not the same, then it is an error. Similarly, if the process' color is 2 and if its parent in the BFS tree has color 0, or if one of its children in the BFS tree has a color different than 2, then it is an error. Upon a color error detection, the color of the process is set to 0.
- Level errors: throughout the algorithm, P makes use of four variables which define a search interval, two for the MinId phase: $P.minlevel$ and $P.minhilevel$, and two for the MaxMinId phase: $P.maxminlevel$ and $P.maxminhilevel$. Depending on the color of the process, the values of $P.minlevel$ and $P.minhilevel$ must fulfill certain conditions, if they do not they are set to k and $k+1$, respectively; the variables $P.maxminlevel$ and $P.maxminhilevel$ are treated similarly.
- Initialization errors: P also makes use of the variables $P.minid$ and $P.maxminid$ to store the minimum ID found in the MinId phase, and the largest $MinId(Q, k)$ found in the MaxMinId phase. If the process has color 0, in order to correctly start the computation of the MinId phase, the values of $P.minid$, $P.minlevel$ and $P.minhilevel$ must be set respectively to $P.id$, 0 and $MinHop(P)$; if they do not have these values, they are corrected. The variables $P.maxminlevel$ and $P.maxminhilevel$ are treated similarly when $P.color = 2$, except that $P.maxminid$ is set to $P.minid$, so that the MaxMinId phase starts correctly.

When all errors have been corrected, no process will ever return to an error state for as long as the algorithm runs without external interference.

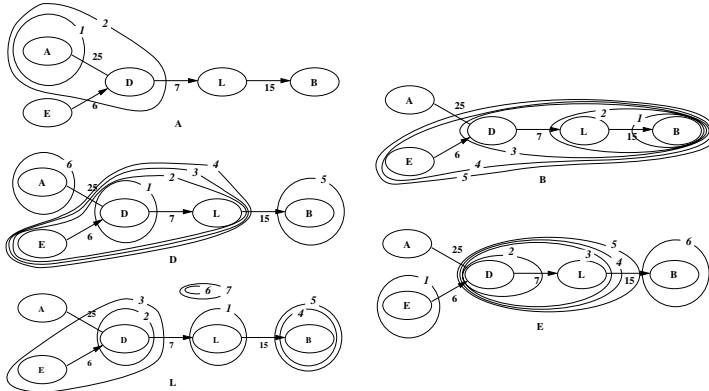


Fig. 1. Evolution of the search interval $P.\text{minlevel} \leq d < P.\text{minhilevel}$ for the example computation Fig. 2

3.5 Building Clusters

The heart of the algorithm is identification of the clusterheads, which consists of two phases, MinId and MaxMinId. We describe only MinId in detail, as MaxMinId is similar. When it ends, if $P.\text{maxminid} = P.\text{id}$, P is a clusterhead.

MinId computes, for each process P , the smallest ID of any process which is within distance k of P . This phase starts when $P.\text{color} = 0$, and ends when $P.\text{minhilevel} = k + 1$. Three steps constitute this phase:

First Step: Synchronization: a color wave starting from the root of the BFS tree sets the color of all processes to 1.

Second Step: At each substep, process P defines a search interval, it gives lower and upper bounds on the distance d up to which P has looked to find the lowest ID: $P.\text{minlevel} \leq d < P.\text{minhilevel}$. The bounds can never decrease in each substep of the algorithm. Initially each process is only able to look no further than itself. Then, a process is able to update its $P.\text{minid}$, $P.\text{minlevel}$ and $P.\text{minhilevel}$ only when no neighbor prevents it from executing, *i.e.*, when P is included in the search interval of one of its neighbors Q , and P has a lower minid value than Q : a neighbor has not finished updating its variables according to its search interval. The levels are increased in accordance with the current levels and the minid of the neighbors: $P.\text{minlevel}$ is set to the minimum $Q.\text{minlevel} + w(P, Q) \leq k$ such that $Q.\text{minid} < P.\text{minid}$, and $P.\text{minhilevel}$ is set to the minimum of all $\min\{Q.\text{minlevel} + w(P, Q), k + 1\}$ if $Q.\text{minid} < P.\text{minid}$, or $\min\{Q.\text{minhilevel} + w(P, Q), k + 1\}$.

Of course, a process cannot directly look at processes which are not its direct neighbors, but the evolution of the search intervals gives time for the information to gradually travel from process to process, thus by reading its neighbors variables, the process will eventually receive the values.

An example of the evolution of the search intervals is given Fig. 11. For example, process E starts by looking at itself, then it looks at D , then for the next three steps it is able to look at D and L , and finally looks at B . It never looks at A , as the distance between E and A is greater than $k = 30$.

Third Step: Once $P.\text{minhilevel} = k+1$, another color wave starts at the bottom of the tree; this wave sets the color of all processes to 2. The processes are now ready to for the MaxMinId phase.

3.6 Time and Space Complexity

The algorithm uses all the variables of SSLE [12] and 11 other variables in each process. SSLE uses $O(\log n)$ space. The internal ID variables can be encoded on $O(\log n)$ space, distance in $O(\log k)$ space, and colors in only 2 bits. Hence, Weighted-Clustering requires $O(\log n + \log k)$ memory per process.

SSLE converges in $O(n)$ rounds, while the clustering module requires $O(n)$ rounds once *Clusterhead_Set* has been correctly computed. MinId and MaxMinId are the most time-consuming, requiring $O(nk)$ rounds each to converge. The total time complexity of the Weighted-Clustering is thus $O(nk)$.

4 An Example Computation

In Fig. 12, we show an example where $k = 30$. In that figure, each oval represents a process P and the numbers on the lines between the ovals represent the weights of the links. To help distinguish IDs from distances, we use letters for IDs. The top letter in the oval representing a process P is $P.\text{id}$. Below that, for subfigures (a) to (g) we show $P.\text{minlevel}$, followed by a colon, followed by $P.\text{minid}$, followed by a colon, followed by $P.\text{minhilevel}$. Below each oval is shown the action the process is enabled to execute (none if the process is disabled). We name A_{minid} the action consisting in updating $P.\text{minlevel}$, $P.\text{minid}$ and $P.\text{minhilevel}$, and A_{hilevel} the action consisting in updating only $P.\text{minhilevel}$. An arrow in the figure from a process P to a process Q indicates that Q prevents P from executing Action A_{minid} . In subfigure (h) we show the final values of $P.\text{maxminlevel}$, followed by a colon, followed by $P.\text{maxminid}$, followed by a colon, followed by $P.\text{maxminhilevel}$. In subfigure (i) we show the final value of $P.\text{dist}$; an arrow from P to Q indicates that $P.\text{parent} = Q.\text{id}$, and a bold oval means that the process is a clusterhead. The dashed line represents the separation between the two final k -clusters.

In Fig. 12(a) to (g), we show synchronous execution of the *MinId* phase. The result would have been the same with an asynchronous execution, but using synchrony makes the example easier to understand.

In each step, if an arrow leaves a process, then this process cannot execute A_{minid} , but can possibly execute A_{hilevel} to update its *minhilevel* variable. At any step, two neighbors cannot both execute Action A_{minid} due to a special condition present in the guard. This prevents miscalculations of *minid*.

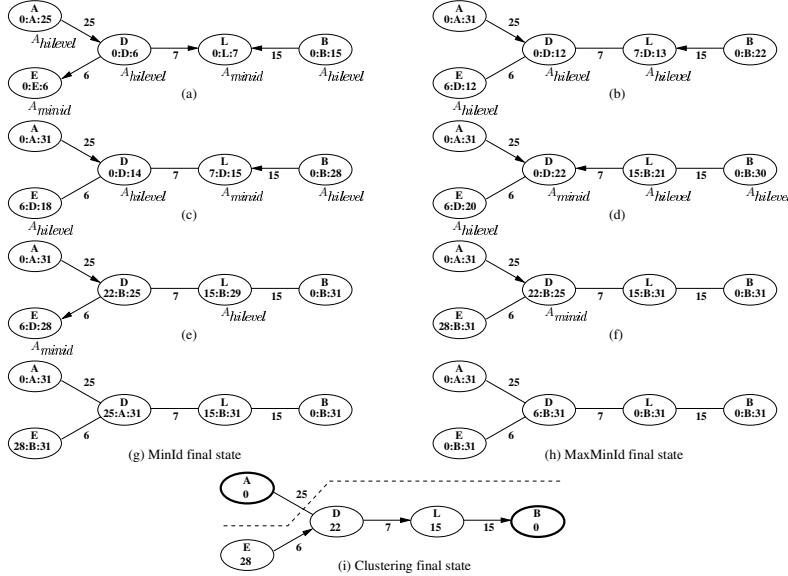


Fig. 2. Example computation of Weighted-Clustering for $k = 30$

Consider the process L . Initially it is enabled to execute Action A_{minid} (subfigure (a)). It will, after the first execution (subfigure (b)), find the value of the smallest ID within a distance of $L.minhilevel = 7$, which is D , and will at the same time update its $minhilevel$ value to $D.minhilevel + w(D, L) = 6 + 7 = 13$. As during this step, D and B have updated their $minhilevel$ value, $L.minhilevel$ is an underestimate of the real $minhilevel$, thus L is now enabled to execute Action $A_{hilevel}$ to correct this value. The idea behind the $minhilevel$ variable, is to prevent the process from choosing a minimum ID at a distance greater than $minhilevel$ before its neighbors have a chance to copy its current value of $minid$, if necessary. Thus a process will not look at the closest minimum ID in terms of number of hops (as could have done process D at the beginning by choosing process A), but will compute the minimum ID within a radius equal to $minhilevel$ around itself (hence process D is only able to choose process A in the final step, even if A is closer than B in terms of number of hops).

The MinId phase halts when $P.minhilevel = k + 1$ for all P (subfigure (g)). In the final step every P knows the process of minimum ID at a distance no greater than k , and $P.minlevel$ holds the distance to this process.

Sometimes, a process P can be elected clusterhead by another process Q without having elected itself clusterhead (this does not appear in our example); P could have the smallest ID of any process within k of Q , but not the smallest ID of any node within k of itself. The *MaxMinId* phase corrects this; it allows the information that a process P was elected clusterhead to flow back to P .

5 Simulations

We designed a simulator to evaluate the performance of our algorithm. In order to verify the results, a sequential version of the algorithm was run, and all simulation results compared to the sequential version results. Thus, we made sure that the returned clustered graph was the correct one. In order to detect when the algorithm becomes stable and has computed the correct clustering, we compared, at each step, the current graph with the previous one; the result was then output only if there was a difference. The stable result is the last graph output once the algorithm has reached an upper bound on the number of rounds (at least two orders of magnitude higher than the theoretical convergence time).

We ran the simulator on random weighted graphs. For each value of k , we ran 10 simulations starting from an arbitrary initial state where the value of each variable of each process was randomly chosen. Each process had a specific computing power so that they could not execute all at the same speed; we set the ratio between the slowest and the fastest process to 1/100. Due to space

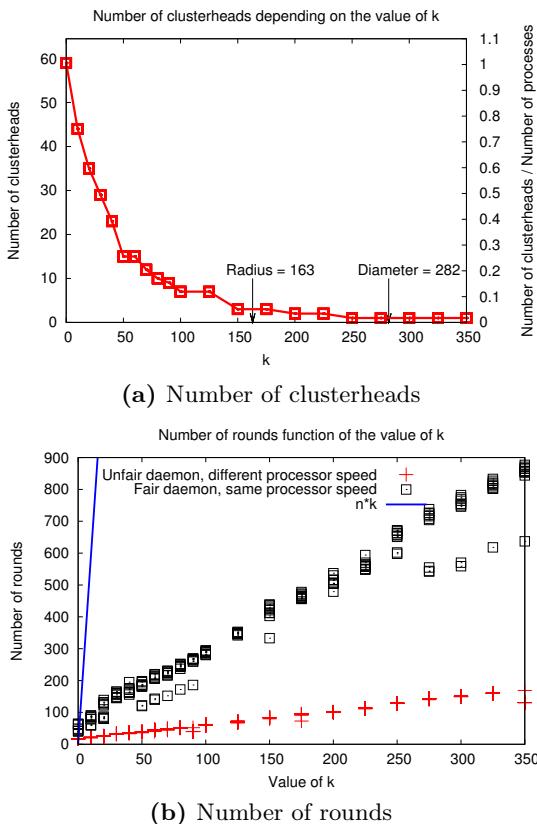


Fig. 3. Simulations results

constraints we cannot show all these results, and present results obtained for one of the random graphs containing 59 nodes, with a diameter equal to 282 (9 hops), links' weights are between 1 and 100, and the degree of the nodes are between 3 and 8 (5.4 on average).

Figure 3a shows the number of clusterheads found for each run and each value of k . As the algorithm returns exactly the same set of clusterheads whatever the initial condition, the results for a given k are all the same. Note that the number of clusterheads decreases as k increases, and even if the algorithm may not find the optimal solution, it gives a clustering far better than a naive $O(1)$ self-stabilizing algorithm which would consist in electing each process a clusterhead. The number of clusterheads quickly decreases as k increases.

Figure 3b shows the number of rounds required to converge. This figure shows two kinds of runs: with an unfair daemon and different computing speed, and with a fair daemon and identical power for all processes. The number of rounds is far lower than the theoretical bound $O(nk)$, even with an unfair daemon.

6 Conclusion

In this article, we present a self-stabilizing asynchronous distributed algorithm for construction of a k -dominating set, and hence a k -clustering, for a given k , for any weighted network. In contrast with previous work, our algorithm deals with an arbitrary metric on the network. The algorithm executes in $O(nk)$ rounds, and requires only $O(\log n + \log k)$ space per process.

In future work, we will attempt to improve the time complexity of the algorithm, and use the message passing model, which is more realistic. We also intend to explore the possibility of using k -clustering to design efficient deployment algorithms for applications on a grid infrastructure.

References

1. Caron, E., Desprez, F.: DIET: A scalable toolbox to build network enabled servers on the grid. *Int. Jour. of HPC Applications* 20(3), 335–352 (2006)
2. YarKhan, A., Dongarra, J., Seymour, K.: GridSolve: The Evolution of Network Enabled Solver. In: Patrick Gaffney, J.C.T.P. (ed.) *Grid-Based Problem Solving Environments: IFIP TC2/WG 2.5 Working Conference on Grid-Based Problem Solving Environments*, Prescott, AZ, July 2006, pp. 215–226. Springer, Heidelberg (2007)
3. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. *Commun. ACM* 17(11), 643–644 (1974)
4. Garey, M.R., Johnson, D.S.: *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York (1979)
5. Amis, A.D., Prakash, R., Vuong, T.H., Huynh, D.T.: Max-min d-cluster formation in wireless ad hoc networks. In: *IEEE INFOCOM*, pp. 32–41 (2000)
6. Spohn, M., Garcia-Luna-Aceves, J.: Bounded-distance multi-clusterhead formation in wireless ad hoc networks. *Ad Hoc Networks* 5, 504–530 (2004)

7. Fernandess, Y., Malkhi, D.: K-clustering in wireless ad hoc networks. In: ACM Workshop on Principles of Mobile Computing POMC 2002, pp. 31–37 (2002)
8. Datta, A.K., Larmore, L.L., Vemula, P.: A self-stabilizing $O(k)$ -time k-clustering algorithm. Computer Journal (2008)
9. Dolev, S.: Self-stabilization. MIT Press, Cambridge (2000)
10. Dolev, S., Israeli, A., Moran, S.: Uniform dynamic self-stabilizing leader election. IEEE Trans. Parallel Distrib. Syst. 8(4), 424–440 (1997)
11. Caron, E., Datta, A.K., Depardon, B., Larmore, L.L.: A self-stabilizing k-clustering algorithm using an arbitrary metric. Technical Report RR2008-31, Laboratoire de l'Informatique du Parallelisme, LIP (2008)
12. Datta, A.K., Larmore, L.L., Vemula, P.: Self-stabilizing leader election in optimal space. In: 10th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS), Detroit, MI (November 2008)

Active Optimistic Message Logging for Reliable Execution of MPI Applications

Thomas Ropars¹ and Christine Morin²

¹ Université de Rennes 1, IRISA, Rennes, France

Thomas.Ropars@irisa.fr

² INRIA Centre Rennes - Bretagne Atlantique, Rennes, France

Christine.Morin@inria.fr

Abstract. To execute MPI applications reliably, fault tolerance mechanisms are needed. Message logging is a well known solution to provide fault tolerance for MPI applications. It has been proved that it can tolerate higher failure rate than coordinated checkpointing. However pessimistic and causal message logging can induce high overhead on failure free execution. In this paper, we present O2P, a new optimistic message logging protocol, based on active optimistic message logging. Contrary to existing optimistic message logging protocols that saves dependency information on reliable storage periodically, O2P logs dependency information as soon as possible to reduce the amount of data piggybacked on application messages. Thus it reduces the overhead of the protocol on failure free execution, making it more scalable and simplifying recovery. O2P is implemented as a module of the Open MPI library. Experiments show that active message logging is promising to improve scalability and performance of optimistic message logging.

1 Introduction

The Mean Time Between Failures of High Performance Computing (HPC) systems is continuously decreasing as the scale of such systems keeps on growing. Efficient fault tolerance mechanisms are needed to enable applications to finish their execution despite frequent failures.

MPI is a widely used paradigm for HPC applications. Transparent fault tolerance solutions are attractive because all the mechanisms needed to handle failures are provided by the library. Thus the application programmer can focus on implementing her application without wasting time with the complex task of failure management. Rollback-recovery techniques are well known techniques to provide transparent fault tolerance for MPI applications [8].

Coordinated checkpointing is the most widely used rollback-recovery technique. Its main drawback is that the failure of one process implies a rollback of all the application processes. Furthermore with coordinated checkpointing all processes are checkpointed and restarted almost at the same time. When stable storage is implemented as a central server, issues due to concurrent access may occur. Message logging has the advantage over coordinated checkpointing protocols to minimize the impact of a failure since they do not require all processes

to rollback in the event of one failure. It can be combined with uncoordinated checkpointing without the risk of domino effect. Additionally message logging is more suitable for applications communicating with the outside world [7]. Previous evaluations of rollback-recovery techniques in an MPI library [14, 3] have shown that message logging protocols, and especially causal message logging protocols, are more scalable than coordinated checkpointing. Furthermore it has been demonstrated that reducing the size of piggybacked data on application messages in causal message logging protocols has a major impact on performance [4]. Optimistic message logging protocols require less information to be piggybacked on messages than causal message logging protocols. Hence optimistic message logging can perform better than causal message logging.

In this paper we present O2P, an optimistic message logging protocol targeting performance and scalability. O2P is a sender-based message logging protocol tolerating multiple concurrent failures. It is based on an innovative active message logging strategy to keep the size of piggybacked data minimal. Implemented as a module of Open MPI [11], it is to our knowledge the first optimistic message logging protocol implemented in a widely used MPI library. Evaluations show that active optimistic message logging is a promising technique to provide an efficient and scalable fault tolerance solution for MPI applications.

The paper is organized as follows. Section 2 presents the related work and motivates this work. The O2P protocol based on active optimistic message logging is described in Section 3. We describe our prototype in Section 4. Experimental results are provided and discussed in this section. Finally, conclusions and future works are detailed in Section 5.

2 Background

2.1 Message Logging Principles

Message logging protocols assume that process execution is piecewise deterministic [21], i.e. the execution of a process is a sequence of deterministic intervals started by a non-deterministic event, a message receipt. This means that starting from the same initial state and delivering the same sequence of messages, two processes inevitably reach the same state.

Determinants [1] describe messages. A determinant is composed of the message payload and a tag. To identify messages, each process numbers the messages it sends with a sender sequence number (*ssn*) and the messages it receives with a receiver sequence number (*rsn*). A message tag is composed of the sender identifier, the *ssn*, the receiver identifier and the *rsn*. Message logging protocols save determinants into stable storage to be able to replay the sequence of messages received by a process in the event of failures. A message is stable when its receiver has saved the corresponding determinant in stable storage. Sender-based message logging [12] is usually used to avoid saving the message payload with the determinant, i.e. the payload is saved in the message sender volatile memory.

The deterministic intervals composing the process execution are called state intervals. A state interval is identified by an index corresponding to the receiver

sequence number of the message starting the interval. Message exchanges create causal dependencies between the state intervals of the application processes. The state intervals are partially ordered by the Lamport's happen-before relation. If some determinants are lost in a process failure, some state intervals of the failed process cannot be recovered. Non-failed processes depending on those lost state intervals become orphan and have to be rolled-back to reach a consistent global state, i.e. a state that could have been seen during failure free execution. Message logging is generally combined with uncoordinated checkpointing. In that case, checkpointing can be seen as a solution to reduce the size of the logs. When a process is checkpointed, the determinants of all the messages it delivered previously can be deleted as soon as it is sure the process will never rollback the checkpointed state. We do not consider checkpoints in the rest of the paper.

The three classes of message logging protocols are: optimistic, pessimistic, and causal. A formal definition of these three classes can be found in [1]. They differ in the way they ensure a consistent global state is reachable after a failure. Pessimistic message logging never creates any orphan processes. Determinants are logged synchronously on stable storage: a process has to wait for the determinants of all the messages it has delivered to be stable before sending a message. Thus in the event of failures, only the failed processes have to restart. Causal message logging also ensures the no-orphan-process condition by piggy-backing on messages the dependency information needed to be able to replay them. On message delivery, the receiving process saves locally those dependency data. Thus when a process fails the information needed to replay the messages is provided by the non-failed processes. Finally, in optimistic message logging determinants are saved asynchronously on stable storage. Application message sending is never delayed but orphan processes might be created. To be able to detect orphan processes, some dependency information is also piggybacked on messages. When a failure occurs all the orphan processes are detected due to the dependency information maintained by each process and are rolled-back to a consistent global state.

2.2 Message Logging Protocols Evaluation

During the last decade, the respective performance of existing rollback-recovery techniques has been compared through experimentation. The MPICH-V project that aims at providing a fault tolerant MPI library provided several results. In [14], they show that message logging tolerates a higher fault frequency than coordinated checkpointing. However pessimistic message logging induces large overhead on failure free execution due to synchronous logging [3]. That is why they implemented a causal message logging protocol [14] that provides better failure free performance since message sending is never delayed. However the causal protocol still induces some overhead because of the size of the dependency information piggybacked on application messages and because of the time needed to compute them before message sending and after message delivery.

Since the size of the piggybacked data is one of the key point in causal message logging performance, they promoted the use of an event logger to reduce this

amount of data [4]. An event logger is an interface to a reliable storage where determinants are saved. The application processes send their determinants to the event logger that sends back an acknowledgement when a determinant is saved. The saved determinants do not need to be piggybacked on the messages anymore and thus the size of the piggybacked data is reduced. The evaluations, made with three causal protocols [9, 13, 14] showed that the use of an event logger has a larger impact on causal message logging performance than any optimization in the computation of the process dependencies. However for high message frequency, causal message logging combined with an event logger still induces a large overhead on failure free execution. Comparing the three classes of message logging protocols on recovery demonstrated that their performance on recovery is very similar [15].

Optimistic message logging requires less information to be piggybacked on messages than causal message logging, as explained in Section 3.1. Hence we have investigated optimistic message logging. Our goal is to take into account the use of an event logger in the design of an optimistic message logging protocol to make it more efficient and scalable than existing optimistic protocols.

3 O2P, an Active Optimistic Message Logging Protocol

O2P is a sender-based optimistic message logging protocol based on active message logging to take advantage of the use of an event logger. In this section, we first briefly explain why optimistic message logging can perform better than causal message logging. Then we describe the system model we consider. Finally, we detail O2P and how it takes advantage of the use of an event logger.

3.1 Optimistic versus Causal Message Logging

Causal message logging combined with an event logger is very similar to optimistic message logging. However optimistic message logging can perform better on failure free execution because it requires less information to be piggybacked on application messages. This is illustrated by the simple scenario described in Figure 11. In this example, messages m_1 and m_2 have not been logged when m_3 is sent. A causal message logging protocol needs to piggyback on m_3 all the information needed to be able to replay this message in the event of a failure, i.e. determinants of m_1 and m_2 must be piggybacked on m_3 . With an optimistic message logging protocol, only the data needed to detect orphan processes are piggybacked, i.e. the dependency to the last non stable state interval. So with optimistic message logging only m_2 determinant needs to piggybacked on m_3 .

3.2 System Model

We consider a distributed application composed of n processes communicating only through messages. Our assumptions about communication channels are the one described in the MPI Standard [10]. Communication channels are FIFO and reliable but there is no bound on message transmission delay. Each process has

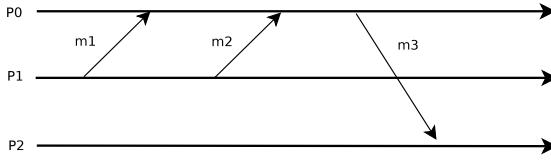


Fig. 1. A simple communication pattern

access to a stable storage server. An event logger act as the interface between the application processes and the stable storage. Information saved in volatile memory is lost in a process crash and only the information saved on stable storage remains accessible. We assume a fail-stop failure model for processes.

3.3 Protocol Description

Optimistic message logging protocols need to track dependencies between processes during failure free execution to be able to detect orphan processes in the event of a failure. Several optimistic message logging protocols are described in the literature varying mainly in the way they track dependencies. To be able to tolerate multiple failures, protocols use either dependency vectors [21, 19] or fault tolerant vector clocks [20, 6]. Damani et al. [5] have demonstrated that dependency vectors have properties similar to Mattern's vector clocks.

O2P uses dependency vectors to track transitive dependencies between processes. A dependency vector is a n entry vector, n being the number of processes in the application. Each entry is composed of an incarnation number and a state interval index. The incarnation number of a process is incremented each time it restarts or rolls-back. Incarnation numbers are used to discard orphan messages coming from old incarnations.

Existing optimistic message logging protocols implement the determinant's logging on stable storage as a periodical task. Determinants of the messages delivered by a process are buffered in the process memory and are periodically saved on stable storage. They don't focus on minimizing piggybacked data size to improve failure free performance. Vectors of size n are piggybacked on application messages to trace dependencies between processes state intervals. This solution is not scalable since the amount of piggybacked data grows with the application size.

However it has been proved that tracking dependency to non-stable state intervals is enough to be able to detect orphan processes after a failure [5]. A process state interval is stable when the determinants of all messages delivered by that process before this state interval have been saved on stable storage. Furthermore we proved in [16] that with tracking dependencies only to non-stable state intervals, it was possible to recover the maximum consistent global state of the application after a failure. O2P takes advantage of these properties to reduce the size of dependency vectors it piggybacks on application messages.

```

Sending a message  $msg$  by process  $p_i$  to process  $p_j$ 
  Piggyback  $DependencyVector_i$  on  $msg$  //  $DependencyVector_i$  is the dependency
  vector of process  $p_i$ 
  Send  $msg$  to  $p_j$ 
  Save  $msg$  in volatile memory

Delivering a message  $(msg, DependencyVector_{msg})$ 
  Get  $(msg, DependencyVector_{msg})$  from the incoming queue
   $s_i \leftarrow s_i + 1$  // Incrementing state interval index
  Deliver  $msg$  to the application
  Update  $DependencyVector_i$  with  $det_{msg}$  //  $det_{msg}$  is the determinant of  $msg$ 
  Update  $DependencyVector_i$  with  $DependencyVector_{msg}$ 
  Send  $det_{msg}$  to the event logger

Upon reception  $Ack(StableVector_{msg})$  on process  $p_i$ 
  for all  $0 \leq k < n$  such that  $StableVector_{msg}[k] \geq DependencyVector_i[k]$  do
     $DependencyVector_i[k] = \perp$ 

```

Fig. 2. O2P failure free protocol

Failure Free Protocol. O2P logs determinants in an active way. Since O2P is a sender based protocol, message payload is not included in the determinants saved in stable storage. A simplified version of the failure free protocol is described in Figure 2. As soon as a message is delivered, its determinant is sent to the event logger to be logged. Process p_i saves in entry j of its dependency vector the last non-stable state interval of p_j its current state interval depends on. This entry is set to \perp if p_i does not depend on any non stable state interval of p_j .

O2P is based on two optimistic assumptions. The first one is the traditional assumption used in optimistic message logging protocols: logging a determinant is fast enough so that the risk of experiencing a failure between message delivery and the log of the corresponding determinant is small. Thus the risk of orphan process creation is low.

The second optimistic assumption we use is that logging a determinant is fast enough so that the probability of having it saved before the next message sending is high. If the second assumption is always valid, all the entries in the dependency vector of a process sending a message are empty.

To take advantage of this, dependency vectors are implemented as described in [18]. Instead of piggybacking the complete dependency vector on every message, only the entries which have changed since the last send to the same process and which are not empty, are sent as piggybacked data. Thus if the second optimistic assumption is always valid, no data has to be piggybacked on the application messages.

Event Logger. The event logger is the interface between the processes and the reliable storage. When it receives a determinant from a process, it saves this determinant and sends back the corresponding acknowledgement. In order to make a process aware of the new determinants saved by other processes, the event logger maintains a n entry vector called $StableVector$. Entry k of the $StableVector$, updated each time the event logger saves a new determinant from process p_k , is the last stable state interval of p_k . This vector is included in the acknowledgements

sent by the event logger. When a process delivers an acknowledgement, it updates its dependency vector according to the *StableVector*. This mechanism contributes to reduce the size of the piggybacked data by avoiding piggybacking information on already stable state intervals.

Recovery Protocol. After a failure, a failed process is restarted from its last checkpoint if any or from the beginning. It gets from the event logger the list of determinants logged before the failure. Then it informs the other processes of its restart and gives its maximum recoverable state interval. When a non-failed process receives a failure notification, it can determine if it is orphan according to its dependency vector. If its dependency vector is empty, it cannot be orphan. The non-orphan processes can continue their execution while the failed and orphan processes interact to find the maximum consistent global state as described in [16].

4 Performance Evaluation

4.1 Implementation Details

We have implemented O2P as a component of the Open MPI library. Open MPI provides a *vampire* Point-to-point Management Layer (PML) that enables to overload the regular PML to execute additional code on every communication request made by the application processes. Thus it enables to implement fault tolerance protocols. O2P is a component of the *vampire* PML framework.

The event logger is a mono-threaded MPI process that can handle asynchronous communications. This process is started separately from the application processes. Application processes connect to the event logger when they start logging determinants.

Different solutions can be used to implement piggyback mechanisms. On each message sending, a new data-type can be created using absolute addresses to attach piggyback data to the application payload. Another solution is to send an additional message with the piggybacked data after the application message. Performance depends on the application and on the MPI implementation [17]. The evaluations we made showed that sending additional messages was more efficient in our case. So it is the solution we adopted for our prototype.

4.2 Experimental Settings

Our experiments are run on a 60-node cluster of Dell PowerEdge 1950 servers. Each node is equipped with an Intel Xeon 5148 LV processor running at 2.33 Ghz, 4 GB of memory and a 160 GB SATA hard drive. An additional Dell PowerEdge 1950 server equipped with an Intel Xeon 5148 LV processor running at 2.33 Ghz, 8 GB of memory and two 300 GB Raid0/SATA hard drives, is used to run the event logger. All nodes, equipped with a Gigabit Ethernet network interface, are linked by a single Cisco 6509 switch and run Linux 2.6.18.

Table 1. Communication rate of the Class A NAS Parallel Benchmarks for 16 processes

	BT	CG	FT	LU	MG	SP
Execution time (in s.)	23.32	0.72	1.54	15.03	0.62	14.73
Messages/second per process	13	256	6	198	108	41

4.3 Results

To evaluate O2P, we used 6 class A applications from the NAS Parallel Benchmark [2], developed by the NASA NAS research center. Table 1 shows the communication frequency per process of these applications. In the experiments, application processes are uniformly distributed over the 60 nodes of the cluster. The measurements we provide are mean values over 5 executions of each test. For the experiments, we compare the active optimistic message logging protocol used by O2P with a *standard* optimistic message logging protocol. This *standard* protocol is not the implementation of any existing optimistic message logging protocol but is a modified version of O2P that doesn't take into account acknowledgments sent by the event logger to update dependency vectors. So processes don't waste time updating their dependency vectors on acknowledgment delivery. But the amount of piggybacked data is also not optimized. We consider that the *standard* protocol behaves like the optimistic protocols described in the literature regarding piggybacked data management.

Size of the Piggybacked Data. To evaluate the impact of active message logging, we measure the amount of data piggybacked on each application message. We run the 6 benchmarks with 4 to 128 processes. Figure 3 presents the mean number of timestamps, i.e. the number of entries in the dependency vector, piggybacked on each message during the execution of the application. We first evaluate this amount of piggybacked data for the *standard* optimistic message logging protocol. This result is the number of timestamps a traditional optimistic message logging protocol would piggyback on application messages. This amount grows linearly with the size of the application, underlining the scalability limit of existing optimistic message logging protocols. The results for the active message logging strategy used by O2P are represented by the curve named *Active*. For applications with low communication rates, i.e. BT and FT, active message logging works perfectly up to 64 processes. Determinants are logged with a very short delay by the event logger. Most of the time, the processes do not depend on any non stable state intervals when they send a message. So they do not have any information to piggyback. Here active optimistic message logging efficiently reduces the amount of piggybacked data. For 128 processes, the amount of data piggybacked is almost the same as for standard optimistic message logging because the event logger is overloaded and does not manage to log the determinants in time. For the benchmarks with a higher communication frequency, this limit is reached earlier.

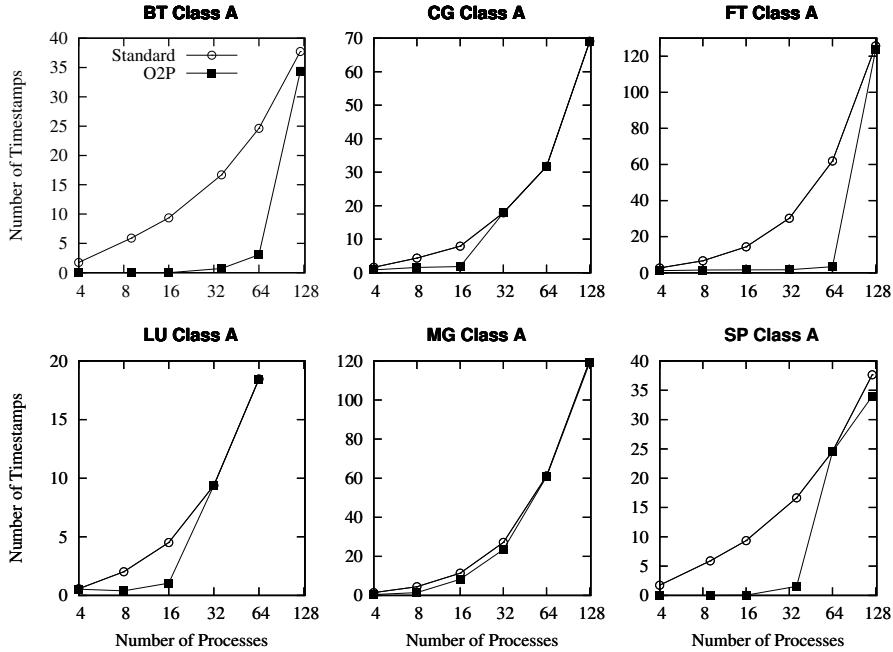


Fig. 3. Amount of data piggybacked on application messages

Failure Free Performance. The overhead induced by O2P on failure free execution for 4 representative benchmarks is summarized in Table 2. Evaluations are made with 16, 32, and 64 processes. Application BT requires a square number of processes. For this application, 36 processes are used instead of 32.

The overhead induced by the *standard* optimistic message logging protocol is compared with O2P. O2P almost always provides better performance than the standard protocol. For the applications with low communication rate, i.e. BT and FT, the overhead induced by O2P is very small. For CG, which has a very high communication rate, the overhead induced by optimistic message logging is significant even if active optimistic message logging contributes to reduce it.

4.4 Discussion

Results show that active optimistic message logging can reduce efficiently the amount of data piggybacked on messages during failure free execution. Thus the performance and scalability of optimistic protocol is improved. Furthermore when active optimistic message logging works well, the mean number of piggybacked timestamps is close to zero. It means that most of the time when a process sends a message it does not depend on any non stable state interval, i.e. the message will never be rolled-back. In this way, active optimistic message logging also reduces the cost of the recovery protocol because it limits the risk of orphan process creation. Furthermore the non-failed processes that do

Table 2. Overhead on failure free execution

N	Protocol	BT	CG	FT	MG
16	Standard	1.14%	12.85%	0.26%	6.80%
	O2P	0.98%	12.01%	0.13%	6.80%
32	Standard	2.49%	21.34%	0.95%	8.57%
	Active	2.22%	21.81%	0.72%	8.57%
64	Standard	3.71%	41.32%	9.23%	15.00%
	Active	3.13%	32.34%	0.00%	15.00%

not depend on any non stable state intervals when a failure occurs, can ignore the failure notification and continue their execution. Future works include the evaluation of the benefits of active optimistic message logging on recovery.

Experiments with large number of processes and with high communication rate applications indicate that the event logger is the bottleneck of our system. Implemented as a mono-thread MPI process, several solutions can be considered to improve it. The event logger could be implemented as a multi-thread MPI process to take advantage of actual multi-core processors. Active replication techniques could also be used to improve both performance and reliability of the event logger. Stable storage can be implemented in a distributed way using the volatile memory of the computation nodes. To make a data stable despite r concurrent failures, it has to be replicated in $r+1$ nodes. Hence we are investigating the implementation of an event logger based on a fault tolerant distributed shared memory such as Juxmem.

5 Conclusion

Message logging is an attractive solution to provide transparent fault tolerance for MPI applications since it can tolerate higher failure rate than coordinated checkpointing [14] and is more suitable for applications communicating with the outside world. It has been proved that making use of a stable storage to reduce the amount of data piggybacked by causal message logging on application messages makes it more efficient and scalable [4]. However causal message logging still induces a large overhead on failure free execution. Optimistic message logging requires less information to be piggybacked on messages than causal message logging. However, due to the risk of orphan process creation, the recovery protocol of optimistic message logging protocols is complex and can be costly. This is why existing optimistic message logging solutions are not widely used.

In this paper, we present O2P, a new optimistic protocol based on active optimistic message logging. Active optimistic message logging aims at overcoming the limits of existing optimistic message logging solutions. Contrary to existing

optimistic protocols that saves determinants periodically, O2P saves determinants as soon as possible to reduce the amount of data piggybacked on messages and reduce the risk of orphan process creation. Thus scalability and performance of optimistic message logging are improved.

O2P is implemented as a module of Open MPI. It is to our knowledge the first optimistic message logging protocol implemented in a widely used MPI library. Experimental results show that active message logging improves the performance and scalability of optimistic message logging. Furthermore it simplifies recovery by reducing the risk of orphan process creation. However our centralized implementation of the event logger, in charge of logging the determinants, is the actual bottleneck of the system. To overcome this limitation, we are investigating a solution based on a distributed shared memory to implement determinant logging. Future works also include the comparison with other message logging protocols implemented in Open MPI.

References

- [1] Alvisi, L., Marzullo, K.: Message Logging: Pessimistic, Optimistic, Causal, and Optimal. *IEEE Transactions on Software Engineering* 24(2), 149–159 (1998)
- [2] Bailey, D., Harris, T., Saphir, W., van der Wilngaart, R., Woo, A., Yarrow, M.: The NAS Parallel Benchmarks 2.0. Technical Report Report NAS-95-020, NASA Ames Research Center (1995)
- [3] Bouteiller, A., Cappello, F., Herault, T., Krawezik, K., Lemarinier, P., Magniette, F.: MPICH-V2: a Fault Tolerant MPI for Volatile Nodes based on Pessimistic Sender Based Message Logging. In: SC 2003: Proceedings of the 2003 ACM/IEEE conference on Supercomputing, Washington, DC, USA, p. 25. IEEE Computer Society Press, Los Alamitos (2003)
- [4] Bouteiller, A., Collin, B., Herault, T., Lemarinier, P., Cappello, F.: Impact of Event Logger on Causal Message Logging Protocols for Fault Tolerant MPI. In: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2005), April 2005, vol. 1, p. 97. IEEE Computer Society Press, Los Alamitos (2005)
- [5] Damani, O.P., Wang, Y.-M., Garg, V.K.: Distributed Recovery with K-optimistic Logging. *Journal of Parallel and Distributed Computing* 63, 1193–1218 (2003)
- [6] Damani, O.P., Garg, V.K.: How to Recover Efficiently and Asynchronously when Optimism Fails. In: International Conference on Distributed Computing systems, pp. 108–115. IEEE Computer Society Press, Los Alamitos (1996)
- [7] Elnozahy, E.N., Zwaenepoel, W.: On The Use And Implementation Of Message Logging. In: 24th International Symposium On Fault-Tolerant Computing (FTCS-24), pp. 298–307. IEEE Computer Society Press, Los Alamitos (1994)
- [8] Elnozahy, E.N.(M.), Alvisi, L., Wang, Y.M., Johnson, D.B.: A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Surveys* 34(3), 375–408 (2002)
- [9] Elnozahy, E.N., Zwaenepoel, W.: Manetho: Transparent Roll Back-Recovery with Low Overhead, Limited Rollback, and Fast Output Commit. *IEEE Transactions on Computers* 41(5), 526–531 (1992)
- [10] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, www mpi-forum.org

- [11] Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In: Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, September 2004, pp. 97–104 (2004)
- [12] Johnson, D.B., Zwaenepoel, W.: Sender-Based Message Logging. In: Digest of Papers: The 17th Annual International Symposium on Fault-Tolerant Computing, pp. 14–19 (1987)
- [13] Lee, B., Park, T., Yeom, H.Y., Cho, Y.: An Efficient Algorithm for Causal Message Logging. In: IEEE Symposium on Reliable Distributed Systems, pp. 19–25. IEEE Computer Society Press, Los Alamitos (1998)
- [14] Lemarinier, P., Bouteiller, A., Herault, T., Krawezik, G., Cappello, F.: Improved message logging versus improved coordinated checkpointing for fault tolerant MPI. In: CLUSTER 2004: Proceedings of the 2004 IEEE International Conference on Cluster Computing, Washington, DC, USA, pp. 115–124. IEEE Computer Society Press, Los Alamitos (2004)
- [15] Rao, S., Alvisi, L., Vin, H.M.: The Cost of Recovery in Message Logging Protocols. In: Symposium on Reliable Distributed Systems, pp. 10–18 (1998)
- [16] Ropars, T., Morin, C.: O2P: An Extremely Optimistic Message Logging Protocol. INRIA Research Report 6357 (November 2007)
- [17] Schulz, M., Bronevetsky, G., de Supinski, B.R.: On the performance of transparent MPI piggyback messages. In: Lastovetsky, A., Kechadi, T., Dongarra, J. (eds.) EuroPVM/MPI 2008. LNCS, vol. 5205, pp. 194–201. Springer, Heidelberg (2008)
- [18] Singhal, M., Kshemkalyani, A.: An efficient implementation of vector clocks. Information Processing Letters 43(1), 47–52 (1992)
- [19] Sistla, A.P., Welch, J.L.: Efficient Distributed Recovery Using Message Logging. In: PODC 1989: Proceedings of the eighth annual ACM Symposium on Principles of distributed computing, pp. 223–238. ACM Press, New York (1989)
- [20] Smith, S.W., Johnson, D.B., Tygar, J.D.: Completely Asynchronous Optimistic Recovery with Minimal Rollbacks. In: FTCS-25: 25th International Symposium on Fault Tolerant Computing Digest of Papers, Pasadena, California, pp. 361–371 (1995)
- [21] Strom, R., Yemini, S.: Optimistic Recovery in Distributed Systems. ACM Transactions on Computing Systems 3(3), 204–226 (1985)

Topic 9

Parallel and Distributed Programming

Introduction

Domenico Talia*, Jason Maassen*, Fabrice Huet*, and Shantenu Jha*

The Parallel and Distributed Programming Topic, Topic 9, is concerned with the development of parallel or distributed applications. Developing such applications is a difficult task and it requires advanced algorithms, realistic modeling, efficient design tools, high performance languages and libraries, and experimental evaluation. This topic provides a forum for presentation of new results and practical experience in this domain. It emphasizes research that facilitates the design and development of correct, high-performance, portable, and scalable parallel program. Related to these central needs, this Topic also includes contributions that assess methods for reusability, performance prediction, large-scale deployment, self-adaptivity, and fault-tolerance. The focus of Topic 9 in 2009 is:

- Parallel-programming paradigms, their evaluation and their comparison.
- Innovative languages for parallel and distributed applications.
- Development and debugging tools for parallel programs.
- Performance analysis, design and implementation of models for parallelism.
- Large-scale experiments and validation of parallel applications.
- Methodological aspects of developing, optimizing and validating parallel programs.
- Domain-specific libraries and languages.
- Parallel-programming productivity and pattern-based parallel programming.
- Abstractions for patterns and usage models on distributed systems.
- Parallel software reusability.

Out of 18 papers submitted to Topic 9, 7 were accepted for presentation at Euro-Par 2009. We thank the authors for submitting papers to this topic and the reviewers for very insightful evaluation of the papers. We specially thank the global chair of topic 9, Domenico Talia, and the vice chair, Shantenu Jha, for coordinating the reviews and for their active participation in the discussions.

* Topic chairs.

A Parallel Numerical Library for UPC

Jorge González-Domínguez¹, María J. Martín¹, Guillermo L. Taboada¹,
Juan Touriño¹, Ramón Doallo¹, and Andrés Gómez²

¹ Computer Architecture Group,
University of A Coruña, Spain

{jgonzalez,mariam,taboada,juan,doallo}@udc.es

² Galicia Supercomputing Center (CESGA),
Santiago de Compostela, Spain
agomez@cesga.es

Abstract. Unified Parallel C (UPC) is a Partitioned Global Address Space (PGAS) language that exhibits high performance and portability on a broad class of shared and distributed memory parallel architectures. This paper describes the design and implementation of a parallel numerical library for UPC built on top of the sequential BLAS routines. The developed library exploits the particularities of the PGAS paradigm, taking into account data locality in order to guarantee a good performance. The library was experimentally validated, demonstrating scalability and efficiency.

Keywords: Parallel computing, PGAS, UPC, Numerical libraries, BLAS.

1 Introduction

The Partitioned Global Address Space (PGAS) programming model provides important productivity advantages over traditional parallel programming models. In the PGAS model all threads share a global address space, just as in the shared memory model. However, this space is logically partitioned among threads, just as in the distributed memory model. In this way, the programmer can exploit data locality in order to increase the performance, at the same time as the shared space facilitates the development of parallel codes. PGAS languages trade off ease of use for efficient data locality exploitation. Over the past several years, the PGAS model has been gaining rising attention. A number of PGAS languages are now ubiquitous, such as Titanium [1], Co-Array Fortran [2] and Unified Parallel C (UPC) [3].

UPC is an extension of standard C for parallel computing. In [4] El-Ghazawi et al. establish, through extensive performance measurements, that UPC can potentially perform at similar levels to those of MPI. Barton et al. [5] further demonstrate that UPC codes can obtain good performance scalability up to thousands of processors with the right support from the compiler and the runtime system. Currently there are commercial and open source based compilers of UPC for nearly all parallel machines.

However, a barrier to a more widespread acceptance of UPC is the lack of library support for algorithm developers. The BLAS (Basic Linear Algebra Subprograms) [6,7] are routines that provide standard building blocks for performing basic vector and matrix operations. They are widely used by scientists and engineers to obtain good levels of performance through an efficient exploitation of the memory hierarchy. The PBLAS (Parallel Basic Linear Algebra Subprograms) [8,9] are a parallel subset of BLAS, which have been developed to assist programmers on distributed memory systems. However, PGAS-based counterparts are not available. In [10] a parallel numerical library for Co-Array Fortran is presented, but this library focuses on the definition of distributed data structures based on an abstract object called object map. It uses Co-Array syntax, embedded in methods associated with distributed objects, for communication between objects based on information in the object map.

This paper presents a library for numerical computation in UPC that improves the programmability and performance of UPC applications. The library contains a relevant subset of the BLAS routines. The developed routines exploit the particularities of the PGAS paradigm, taking into account data locality in order to guarantee a good performance. Besides, the routines use internally BLAS calls to carry out the sequential computation inside each thread. The library was experimentally validated on the HP Finis Terrae supercomputer [11].

The rest of the paper is organized as follows. Section 2 describes the library design: types of functions (shared and private), syntax and main characteristics. Section 3 explains the data distributions used for the private version of the routines. Section 4 presents the experimental results obtained on the Finis Terrae supercomputer. Finally, conclusions are discussed in Section 5.

2 Library Design

Each one of the selected BLAS routines has two UPC versions, a *shared* and a *private* one. In the shared version the data distributions are provided by the user through the use of shared arrays with a specific `block_size` (that is, the blocking factor or number of consecutive elements with affinity to the same thread). In the private version the input data are private. In this case the data are transparently distributed by the library. Thus, programmers can use the library independently of the memory space where the data are stored, avoiding to make tedious and error-prone distributions. Table 1 lists all the implemented routines, giving a total of 112 different routines ($14 \times 2 \times 4$).

All the routines return a local integer error value which refers only to each thread execution. If the programmer needs to be sure that no error happened in any thread, the checking must be made by himself using the local error values. This is a usual practice in parallel libraries to avoid unnecessary synchronization overheads.

The developed routines do not implement the numerical operations (e.g. dot product, matrix-vector product, etc.) but they call internally to BLAS routines to perform the sequential computations in each thread. Thus, the UPC BLAS

Table 1. UPC BLAS routines. All the routines follow the naming convention: `upc_blas_[p]Tblasname`. Where the character "p" indicates private function, that is, the input arrays are private; character "T" indicates the type of data (i=integer; l=long; f=float; d=double); and `blasname` is the name of the routine in the sequential BLAS library.

BLAS level	Tblasname	Action
BLAS1	Tcopy	Copies a vector
	Tswap	Swaps the elements of two vectors
	Tscal	Scales a vector by a scalar
	Taxpy	Updates a vector using another one: $y = \alpha * x + y$
	Tdot	Dot product
	Tnrm2	Euclidean norm
	Tasum	Sums the absolute value of the elements of a vector
	iTamax	Finds the index with the maximum value
	iTamin	Finds the index with the minimum value
BLAS2	Tgemv	Matrix-vector product
	Ttrsv	Solves a triangular system of equations
	Tger	Outer product
BLAS3	Tgemm	Matrix-matrix product
	Ttrsm	Solves a block of triangular systems of equations

routines act as an interface to distribute data and synchronize the calls to the BLAS routines in an efficient and transparent way.

2.1 Shared Routines

The UPC language has two standard libraries: the collective library (integrated into the language specification, v1.2 [3]) and the I/O library [12]. The shared version of the numerical routines follows the syntax style of the collective UPC functions. Using the same syntax style as the UPC collective routines eases the learning process to the UPC users. For instance, the syntax of the UPC dot product routine with shared data is:

```
int upc_blas_ddot(const int block_size, const int size, shared
    const double *x, shared const double *y, shared double *dst);
```

being `x` and `y` the source vectors of length `size`, and `dst` the pointer to shared memory where the dot product result will be written; `block_size` is the blocking factor of the source vectors. This function treats `x` and `y` pointers as if they had type `shared [block_size] double[size]`.

In the case of BLAS2 and BLAS3 routines, an additional parameter (`dimmDist`) is needed to indicate the dimension used for the matrix distribution because shared arrays in UPC can only be distributed in one dimension. The UPC routine to solve a triangular system of equations is used to illustrate this issue:

```
int upc blas_dtrsv(const UPC_PBLAS_TRANSPOSE transpose, const
    UPC_PBLAS_UPLO uplo, const UPC_PBLAS_DIAG diag, const
    UPC_PBLAS_DIMMDIST dimmDist, const int block_size, const
    int n, shared const double *M, shared double *x);
```

being **M** the source matrix and **x** the source and result vector; **block_size** is the blocking factor of the source matrix; **n** the number of rows and columns of the matrix; **transpose**, **uplo** and **diag** enumerated values which determine the characteristics of the source matrix (transpose/non-transpose, upper/lower triangular, elements of the diagonal equal to one or not); **dimmDist** is another enumerated value to indicate if the source matrix is distributed by rows or columns. The meaning of the **block_size** parameter depends on the **dimmDist** value. If the source matrix is distributed by rows (**dimmDist=upc_pblas_rowDist**), **block_size** is the number of consecutive rows with affinity to the same thread. In this case, this function treats pointer **M** as if it had type **shared[block_size*n] double[n*n]**. Otherwise, if the source matrix is distributed by columns (**dimmDist=upc_pblas_colDist**), **block_size** is the number of consecutive columns with affinity to the same thread. In this case, this function treats pointer **M** as if it had type **shared[block_size] double[n*n]**.

As mentioned before, in the shared version of the BLAS2 and BLAS3 routines it is not possible to distribute the matrices by 2D blocks (as the UPC syntax does not allow it), which can be a limiting factor for some kinds of parallel computations. To solve this problem, an application layer with support for various dense matrix decomposition strategies is presented in [13]. A detailed discussion about its application to a particular class of algorithms is also included. However, such a support layer still requires considerable development to become useful for a generic numerical problem. In [14] an extension to the UPC language that allows the programmer to block shared arrays in multiple dimensions is proposed. This extension is not currently part of the standard.

2.2 Private Routines

The private version of the routines does not store the input data in a shared array distributed among the threads, but data are completely stored in the private memory of one or more threads. Its syntax is similar to the shared one, but the **block_size** parameter is omitted as in this case the data distribution is internally applied by the library. For instance, the syntax for the dot routine is:

```
int upc blas_pddot(const int size, const int src_thread, const
    double *x, const double *y, const int dst_thread, double *dst);
```

being **x** and **y** the source vectors of length **size**; **dst** the pointer to private memory where the dot product result will be written; and **src_thread/dst_thread** the rank number of the thread (0,1,...**THREADS**-1, being **THREADS** the total number of threads in the UPC execution) where the input/result is stored. If

`src_thread=THREADS`, the input is replicated in all the threads. Similarly, if `dst_thread=THREADS` the result will be replicated to all the threads.

The data distributions used internally by the private routines will be explained in the next section. Unlike the shared version of the routines, in the private one a 2D block distribution for the matrices can be manually built.

2.3 Optimization Techniques

There exist a number of known optimization techniques that improve the efficiency and performance of the UPC codes. The following optimizations have been applied to the implementation of the routines whenever possible:

- Space privatization: When dealing with local shared data, they are accessed through standard C pointers instead of using UPC pointers to shared memory. Shared pointers often require more storage and are more costly to dereference. Experimental measurements in [4] have shown that the use of shared pointers increases execution times by up to three orders of magnitude. For instance, space privatization is widely used in our routines when a thread needs to access only to the elements of a shared array with affinity to that thread.
- Aggregation of remote shared memory accesses: This can be established through block copies, using `upc_memget` and `upc_memput` on remote blocks of data required by a thread, as will be shown in the next section.
- Overlapping of remote memory accesses with computation: It was achieved by the usage of split-phase barriers. For instance, these barriers are used in the triangular solver routines to overlap the local computation of each thread with the communication of partial results to the other threads (see Section 3.3).

3 Data Distributions for the Private Routines

UPC local accesses can be one or two orders of magnitude faster than UPC remote accesses. Thus, all the private functions have been implemented using data distributions that minimize accesses to remote memory in a transparent way to the user.

As mentioned in the previous section, if the parameter `dst_thread=THREADS`, the piece of result obtained by each thread is replicated to all the other ones. To do this, two different options were considered:

- Each thread copies its piece of result into `THREADS` shared arrays, each one with affinity to a different thread. This leads to `THREADS-1` remote accesses for each thread, that is, $THREADS \times (THREADS - 1)$ accesses.
- Each thread first copies its piece of result into a shared array with affinity to thread 0. Then, each thread copies the whole result from thread 0 to its private memory. In this case the number of remote accesses is only $2 \times (THREADS-1)$, although in half of these accesses (the copies in private memory) the amount of data transferred is greater than in the first option.

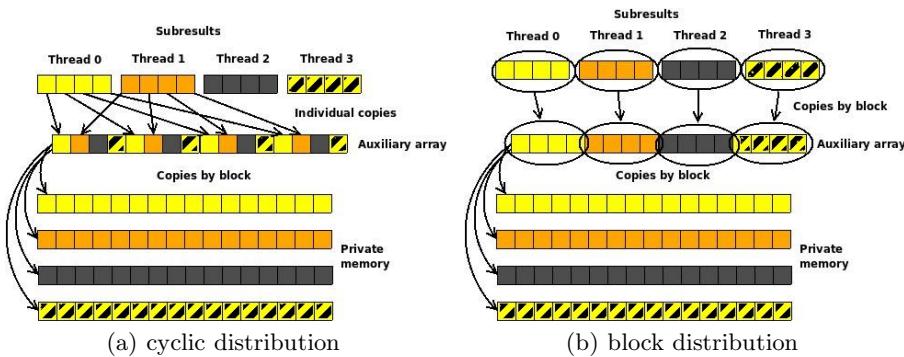


Fig. 1. Data movement using a block and a cyclic distribution

A preliminary performance evaluation of these two options (using the benchmarks of Section 4) has shown that the second one achieves the highest performance, especially on distributed memory systems.

Due to the similarity among the algorithms used in each BLAS level, the chosen distribution has been the same for all the routines inside the same level (except for the triangular solver).

3.1 BLAS1 Distributions

UPC BLAS1 consists of routines that operate with one or more dense vectors. Figure 1 shows the copy of the result in all the threads for a cyclic and a block distribution using the procedure described above (the auxiliary array is a shared array with affinity to thread 0). The block distribution was chosen because all the copies can be carried out in block, allowing the use of the `upc_memput()` and `upc_memget()` functions.

3.2 BLAS2 Distributions

This level contains matrix-vector operations. A matrix can be distributed by rows, by columns or by 2D blocks. The matrix-vector product will be used as an example to explain the chosen distribution.

Figure 2 shows the routine behavior for a column distribution. In order to compute the i -th element of the result, all the i -th values of the subresults should be added. Each of these adds is performed using the `upc_all_reduceT` function (with `upc_op_t op=UPC_ADD`) from the collective library. Other languages allow to compute the set of all the reduction operations in an efficient way using a unique collective function (e.g. `MPI_Reduce`). However, UPC has not such a function, at least currently.

The row distribution is shown in Figure 3. In this case each thread computes a piece of the final result (subresults in the figure). These subresults only must be copied in an auxiliary array in the right position, and no reduction operation is necessary.

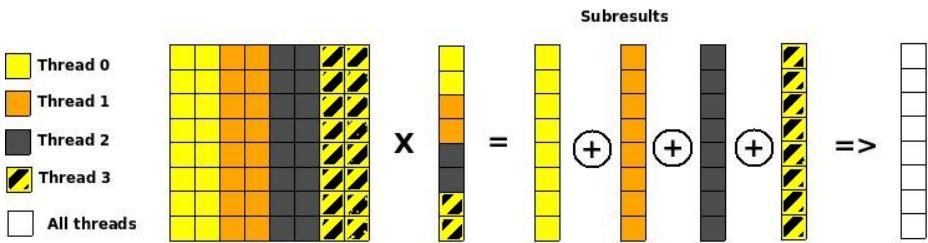


Fig. 2. Matrix-vector product using a column distribution for the matrix

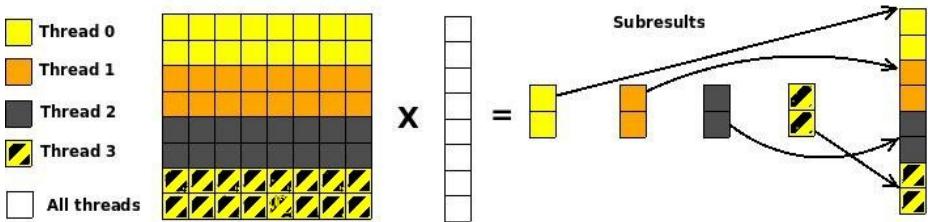


Fig. 3. Matrix-vector product using a row distribution for the matrix

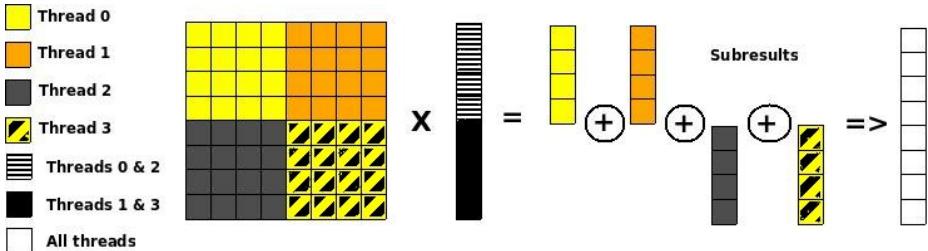


Fig. 4. Matrix-vector product using a 2D block distribution for the matrix

Finally, Figure 4 shows a 2D block distribution. This is a good alternative for cache locality exploitation. This option involves a reduction operation for each row. Each one of these reductions must be performed by all the threads (although not all the threads have elements for all the rows) because UPC does not allow to use a subset of threads in collective functions.

Experimental results showed that the row distribution is the best option as no collective operation is needed. In [15] Nishtala et al. propose extensions to the UPC collective library in order to allow subsets (or teams) of threads to perform a collective function together. If these extensions were included in the UPC specification the column and 2D block distributions should be reconsidered.

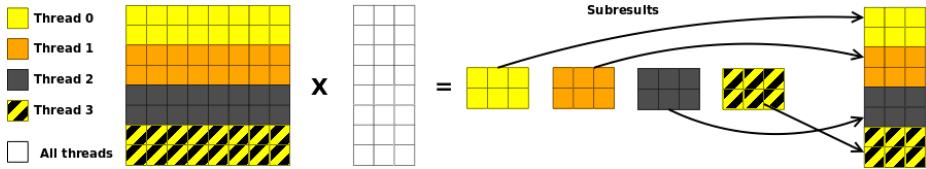


Fig. 5. Matrix-matrix product using a row distribution for the matrix

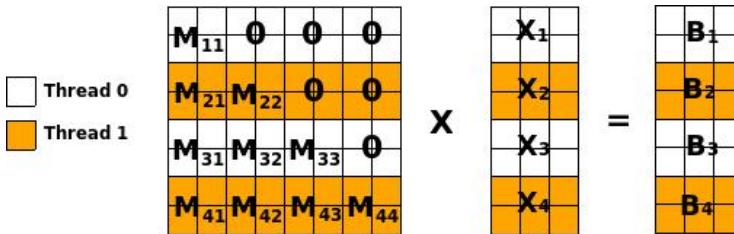


Fig. 6. Matrix distribution for the `upc_blas_pTtrsm` routine

3.3 BLAS3 Distributions

This level contains matrix-matrix operations. Once again, a matrix can be distributed by rows, by columns or by 2D blocks. Actually, the advantages and disadvantages of each distribution are the same discussed in the previous section for BLAS2 routines, so the row distribution was selected again. Figure 5 shows the routine behavior for the matrix-matrix product using this row distribution.

Regarding the matrix distribution, the routines to solve triangular systems of equations (`Ttrsv` and `Ttrsm`) are a special case. In these routines, the rows of the matrix are not distributed in a block way but in a block-cyclic one in order to increase the parallelism and balance the load. Figure 6 shows an example for the `Ttrsm` routine with two threads, two blocks per thread and a source matrix with the following options (see the syntax of the similar `Ttrsv` routine in Section 2.1): non-transpose (`transpose=upc_pblas_noTrans`), lower triangular (`uplo=upc_pblas_lower`), and not all the main diagonal elements equal to one (`diag=upc_pblas_nonUnit`). The triangular matrix is logically divided in square blocks M_{ij} . These blocks are triangular submatrices if $i = j$, square submatrices if $i > j$, and null submatrices if $i < j$.

The algorithm used by this routine is shown in Figure 7. The triangular system can be computed as a sequence of triangular solutions (`Ttrsm`) and matrix-matrix multiplications (`Tgemm`). Note that all operations between two synchronizations can be performed in parallel. The more blocks the matrix is divided in, the more computations can be simultaneously performed, but the more synchronizations are needed too. Experimental measurements have been made to find the block size with the best trade-off between parallelism and synchronization overhead. For our target supercomputer (see Section 4) and large matrices ($n > 2000$) this value is approximately $1000/\text{THREADS}$.

1	$X_1 \leftarrow \text{Solve } M_{11} * X_1 = B_1$	BLAS <code>Ttrsm()</code>
2	---	
3	$B_2 \leftarrow B_2 - M_{21} * X_1$	BLAS <code>Tgemm()</code>
4	$B_3 \leftarrow B_3 - M_{31} * X_1$	BLAS <code>Tgemm()</code>
5	$B_4 \leftarrow B_4 - M_{41} * X_1$	BLAS <code>Tgemm()</code>
6	$X_2 \leftarrow \text{Solve } M_{22} * X_2 = B_2$	BLAS <code>Ttrsm()</code>
7	---	
8	$B_3 \leftarrow B_3 - M_{32} * X_2$	BLAS <code>Tgemm()</code>
9	...	

Fig. 7. UPC BLAS `Ttrsm` algorithm

4 Experimental Results

In order to evaluate the performance of the library, different benchmarks were run on the Finis Terrae supercomputer installed at the Galicia Supercomputing Center (CESGA), ranked #427 in November 2008 TOP500 list (14 TFlops) [11]. It consists of 142 HP RX7640 nodes, each of them with 16 IA64 Itanium 2 Montvale cores at 1.6 Ghz distributed in two cells (8 cores per cell), 128 GB of memory and a dual 4X Infiniband port (16 Gbps of theoretical effective bandwidth).

As UPC programs can be run on shared or distributed memory, the results were obtained using the hybrid memory configuration (shared memory for intra-node communication and Infiniband transfers for inter-node communication). This hybrid architecture is very interesting for PGAS languages, allowing the locality exploitation of threads running in the same node, as well as enhancing scalability through the use of distributed memory.

Among all the possible hybrid configurations, four threads per node, two per cell, was chosen. The use of this configuration represents a good trade-off between shared-memory access performance and scalable access through the Infiniband network. The compiler used is the Berkeley UPC 2.6.0 [16], and the Intel Math Kernel Library (MKL) 9.1 [17], a highly tuned BLAS library for Itanium cores, is the underlying sequential BLAS library.

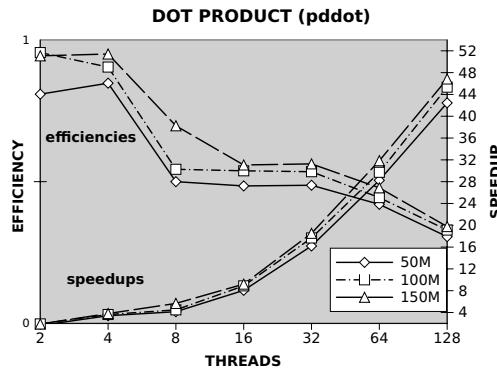
The times measured in all the experiments were obtained for the private version of the routines with the inputs (matrices/vectors) initially replicated in all threads (`src_thread=THREADS`), and results stored in thread 0 (`dst_thread=0`). Results for one core have been obtained with the sequential MKL version.

Results taken from shared routines are not presented because they are very dependent on the data distribution chosen by the user. If programmers choose the best distribution (that is, the same used in the private versions and explained in Section 3), times and speedups obtained would be similar to those of the private counterparts.

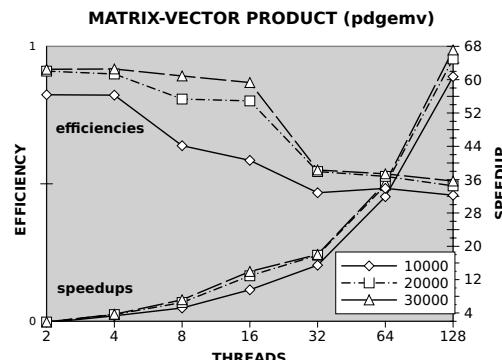
Table 2 and Figure 8 show the execution times, speedups and efficiencies obtained for different vector sizes and number of threads of the `pddot` routine, an example of BLAS1 level routine with arrays of doubles stored in private memory. The size of the arrays is measured in millions of elements. Despite

Table 2. BLAS1 pddot times (ms)

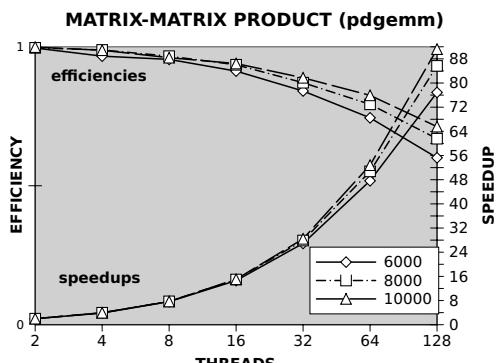
DOT PRODUCT (pddot)			
Size	50M	100M	150M
Thr.	50M	100M	150M
1	147,59	317,28	496,37
2	90,47	165,77	262,37
4	43,25	87,38	130,34
8	35,58	70,75	87,60
16	18,30	35,70	53,94
32	9,11	17,95	26,80
64	5,22	10,68	15,59
128	3,48	7,00	10,60

**Fig. 8.** BLAS1 pddot efficiencies/speedups**Table 3.** BLAS2 pdgemv times (ms)

MATRIX-VECTOR PRODUCT (pdgemv)			
Size	10000	20000	30000
Thr.	10000	20000	30000
1	145,08	692,08	1.424,7
2	87,50	379,24	775,11
4	43,82	191,75	387,12
8	27,93	106,30	198,88
16	15,18	53,58	102,09
32	9,38	38,76	79,01
64	4,55	19,99	40,48
128	2,39	10,65	21,23

**Fig. 9.** BLAS2 pdgemv efficiencies/speedups**Table 4.** BLAS3 pdgemm times (s)

MATRIX-MATRIX PRODUCT (pdgemm)			
Size	6000	8000	10000
Thr.	6000	8000	10000
1	68,88	164,39	319,20
2	34,60	82,52	159,81
4	17,82	41,57	80,82
8	9,02	21,27	41,53
16	4,72	10,99	21,23
32	2,56	5,90	11,23
64	1,45	3,24	6,04
128	0,896	1,92	3,50

**Fig. 10.** BLAS3 pdgemm efficiencies/speedups

computations are very short (in the order of milliseconds), this function scales reasonably well. Only the step from four to eight threads decrease the slope of the curve, as eight threads is the first configuration where not only shared memory but also Infiniband communications are used.

Times, speedups and efficiencies for the matrix-vector product (`pdgemv`, an example of BLAS2 level) are shown in Table 3 and Figure 9. Square matrices are used; the size represents the number of rows and columns. As can be observed, speedups are quite high despite the times obtained from the executions are very short.

Finally, Table 4 and Figure 10 show times (in seconds), speedups and efficiencies obtained from the execution of a BLAS3 routine, the matrix-matrix product (`dgemm`). Input matrices are also square. Speedups are higher in this function because of the large computational cost of its sequential version so that the UPC version benefits from the high ratio computation/communication time.

5 Conclusions

To our knowledge, this is the first parallel numerical library developed for UPC. Numerical libraries improve performance and programmability of scientific and engineering applications. Up to now, in order to use BLAS libraries, parallel programmers have to resort to MPI or OpenMP. With the library presented in this paper, UPC programmers can also benefit from these highly efficient numerical libraries, which allows for broader acceptance of this language.

The library implemented allows to use both private and shared data. In the first case the library decides in a transparent way the best data distribution for each routine. In the second one the library works with the data distribution provided by the user. In both cases UPC optimization techniques, such as privatization or bulk data movements, are applied in order to improve the performance.

BLAS library implementations have evolved over about two decades and are therefore extremely mature both in terms of stability and efficiency for a wide variety of architectures. Thus, the sequential BLAS library is embedded in the body of the corresponding UPC routines. Using sequential BLAS not only improves efficiency, but it also allows to incorporate automatically the new BLAS versions as soon as available.

The proposed library has been experimentally tested, demonstrating scalability and efficiency. The experiments were performed on a multicore supercomputer to show the adequacy of the library to hybrid architectures (shared/distributed memory).

As ongoing work we are currently developing efficient UPC sparse BLAS routines for parallel sparse matrix computations.

Acknowledgments

This work was funded by the Ministry of Science and Innovation of Spain under Project TIN2007-67537-C03-02. We gratefully thank CESGA (Galicia Supercomputing Center, Santiago de Compostela, Spain) for providing access to the Finis Terrae supercomputer.

References

1. Titanium Project Home Page, <http://titanium.cs.berkeley.edu/> (last visit: May 2009)
2. Co-Array Fortran, <http://www.co-array.org/> (last visit: May 2009)
3. UPC Consortium: UPC Language Specifications, v1.2. (2005), http://upc.lbl.gov/docs/user/upc_spec_1.2.pdf
4. El-Ghazawi, T., Cantonnet, F.: UPC Performance and Potential: a NPB Experimental Study. In: Proc. 14th ACM/IEEE Conf. on Supercomputing (SC 2002), Baltimore, MD, USA, pp. 1–26 (2002)
5. Barton, C., Cascajal, C., Almási, G., Zheng, Y., Farreras, M., Chatterje, S., Amaral, J.N.: Shared Memory Programming for Large Scale Machines. In: Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2006), Ottawa, Ontario, Canada, pp. 108–117 (2006)
6. BLAS Home Page, <http://www.netlib.org/blas/> (last visit: May 2009)
7. Dongarra, J.J., Croz, J.D., Hammarling, S., Hanson, R.J.: An Extended Set of FORTRAN Basic Linear Algebra Subprograms. ACM Transactions on Mathematical Software 14(1), 1–17 (1988)
8. PBLAS Home Page, <http://www.netlib.org/scalapack/pblasqref.html> (last visit: May 2009)
9. Choi, J., Dongarra, J.J., Ostrouchov, S., Petitet, A., Walker, D., Clinton Whaley, R.: A Proposal for a Set of Parallel Basic Linear Algebra Subprograms. In: Waśniewski, J., Madsen, K., Dongarra, J. (eds.) PARA 1995. LNCS, vol. 1041, pp. 107–114. Springer, Heidelberg (1996)
10. Numrich, R.W.: A Parallel Numerical Library for Co-array Fortran. In: Wyrzykowski, R., Dongarra, J., Meyer, N., Waśniewski, J. (eds.) PPAM 2005. LNCS, vol. 3911, pp. 960–969. Springer, Heidelberg (2006)
11. Finis Terrae Supercomputer, <http://www.top500.org/system/9500> (last visit: May 2009)
12. El-Ghazawi, T., Cantonnet, F., Saha, P., Thakur, R., Ross, R., Bonachea, D.: UPC-IO: A Parallel I/O API for UPC v1.0 (2004), <http://upc.gwu.edu/docs/UPC-IOv1.0.pdf>
13. Brown, J.L., Wen, Z.: Toward an Application Support Layer: Numerical Computation in Unified Parallel C. In: Wyrzykowski, R., Dongarra, J., Meyer, N., Waśniewski, J. (eds.) PPAM 2005. LNCS, vol. 3911, pp. 912–919. Springer, Heidelberg (2006)
14. Barton, C., Cașcaval, C., Almasi, G., Garg, R., Amaral, J.N., Farreras, M.: Multidimensional Blocking in UPC. In: Adve, V., Garzarán, M.J., Petersen, P. (eds.) LCPC 2007. LNCS, vol. 5234, pp. 47–62. Springer, Heidelberg (2008)
15. Nishtala, R., Almási, G., Cașcaval, C.: Performance without Pain = Productivity: Data Layout and Collective Communication in UPC. In: Proc. 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP 2008), Salt Lake City, UT, USA, pp. 99–110 (2008)
16. Berkeley UPC Project, <http://upc.lbl.gov> (last visit: May 2009)
17. Intel Math Kernel Library, <http://www.intel.com/cd/software/products/asmo-na/eng/307757.htm> (last visit: May 2009)

A Multilevel Parallelization Framework for High-Order Stencil Computations

Hikmet Dursun, Ken-ichi Nomura, Liu Peng, Richard Seymour,
Weiqiang Wang, Rajiv K. Kalia, Aiichiro Nakano, and Priya Vashishta

Collaboratory for Advanced Computing and Simulations,

Department of Computer Science,

Department of Physics & Astronomy,

Department of Chemical Engineering & Materials Science,

University of Southern California, Los Angeles, CA 90089-0242, USA

{hdursun, knomura, liupeng, rseymour, wangweiq, rkalia, anakano,
priyav}@usc.edu

Abstract. Stencil based computation on structured grids is a common kernel to broad scientific applications. The order of stencils increases with the required precision, and it is a challenge to optimize such high-order stencils on multicore architectures. Here, we propose a multilevel parallelization framework that combines: (1) inter-node parallelism by spatial decomposition; (2) intra-chip parallelism through multithreading; and (3) data-level parallelism via single-instruction multiple-data (SIMD) techniques. The framework is applied to a 6th order stencil based seismic wave propagation code on a suite of multicore architectures. Strong-scaling scalability tests exhibit superlinear speedup due to increasing cache capacity on Intel Harpertown and AMD Barcelona based clusters, whereas weak-scaling parallel efficiency is 0.92 on 65,536 BlueGene/P processors. Multithreading+SIMD optimizations achieve 7.85-fold speedup on a dual quad-core Intel Clovertown, and the data-level parallel efficiency is found to depend on the stencil order.

Keywords: Stencil computation, multithreading, single instruction multiple data parallelism, message passing, spatial decomposition.

1 Introduction

Design complexity and cooling difficulties in high-speed single-core chips have forced chip makers to adopt a multicore strategy in designing heterogeneous hardware architectures [1-3]. The shift in architectural design has provided incentives for the high-performance computing (HPC) community to develop a variety of programming paradigms that maximally utilize underlying hardware for broad computational applications [4].

A common core computational kernel used in a variety of scientific and engineering applications is stencil computation (SC). Extensive efforts have been made to optimize SC on multicore platforms with the main focus on low-order SC. For example, Williams et al. [5] have optimized a lattice Boltzmann application on

leading multicore platforms, including Intel Itanium2, Sun Niagara2 and STI Cell. Datta et al. have recently performed comprehensive SC optimization and auto-tuning with both cache-aware and cache-oblivious approaches on a variety of state-of-the-art architectures, including NVIDIA GTX280, etc [6]. Other approaches to SC optimization include methods such as tiling [7] and iteration skewing (if the iteration structure allows it) [8-10]. Due to the importance of high-order SC in the broad applications and the wide landscape of multicore architectures as mentioned above, it is desirable to develop a unified parallelization framework and perform systematic performance optimization for the high-order SC on various multicore architectures.

In this paper, we propose a multilevel parallelization framework addressing high-order stencil computations. Our framework combines: (1) inter-node parallelism by spatial decomposition; (2) intra-chip parallelism through multithreading; and (3) data-level parallelism via single-instruction multiple-data (SIMD) techniques. We test our generic approach with a 6th order stencil based seismic wave propagation code on a suite of multicore architectures. Strong-scaling scalability tests exhibit superlinear speedup due to increasing cache capacity on Intel Harpertown and AMD Barcelona based clusters, whereas weak-scaling parallel efficiency is 0.92 on 65,536 BlueGene/P processors. Our intra-core optimizations combine Multithreading+ SIMDization approaches to achieve 7.85-fold speedup on a dual quad-core Intel Clovertown. We also explore extended precision high order stencil computations at 7th and 8th orders to show dependency of data-level parallel efficiency on the stencil order.

This paper is organized as follows: An overview of the stencil problem is given in Section 2 together with the description of experimental application. Section 3 presents the parallelization framework and details its levels. Section 4 describes the suite of experimental platforms, with details on the input decks used and the methodology used to undertake inter-node and intra-node scalability analysis. Conclusions from this work are discussed in Section 5.

2 High-Order Stencil Application

This section introduces the general concept of high-order stencil based computation as well as details of our experimental application, i.e., seismic wave propagation code.

2.1 Stencil Computation

Stencil computation (SC) is at the heart of a wide range of scientific and engineering applications. A number of benchmark suites, such as PARKBENCH [11], NAS Parallel Benchmarks [12], SPEC [13] and HPFBench [14] include stencil computations to evaluate performance characteristics of HPC clusters. Implementation of special purpose stencil compilers highlights the common use of stencil computation based methods [15]. Other examples to applications employing SC include thermodynamically and mechanically driven flow simulations, e.g. oceanic circulation modeling [16], neighbor pixel based computations, e.g. multimedia/image-processing [17], quantum dynamics [18] and computational electromagnetics [19] software using the finite-difference time-domain method, Jacobi and multigrid solvers [10].

SC involves a field that assigns values $v_t(\mathbf{r})$ to a set of discrete grid points $\Omega = \{\mathbf{r}\}$, for the set of simulation time steps $T = \{t\}$. SC routine sweeps over Ω iteratively to update $v_t(\mathbf{r})$ using a numerical approximation technique as a function of the values of the neighboring nodal set including the node of interest, $\Omega' = \{\mathbf{r}' \mid \mathbf{r}' \in \text{neighbor}(\mathbf{r})\}$ which is determined by the stencil geometry. The pseudocode below shows a naïve stencil computation:

```
for  $\forall t \in T$ 
  for  $\forall \mathbf{r} \in \Omega$ 
     $v_{t+1}(\mathbf{r}) \leftarrow f(\{v_t(\mathbf{r}') \mid \mathbf{r}' \in \text{neighbor}(\mathbf{r})\})$ 
```

where f is the mapping function and v_t is the scalar field at time step t . SC may be classified according to the geometric arrangement of the nodal group $\text{neighbor}(\mathbf{r})$ as follows: First, the order of a stencil is defined as the distance between the grid point of interest, \mathbf{r} , and the farthest grid point in $\text{neighbor}(\mathbf{r})$ along a certain axis. (In a finite-difference application, the order increases with required level of precision.) Second, we define the size of a stencil as the cardinality $|\{\mathbf{r}' \mid \mathbf{r}' \in \text{neighbor}(\mathbf{r})\}|$, i.e., the number of grid points involved in each stencil iteration. Third, we define the footprint of a stencil by the cardinality of minimum bounding orthorhombic volume, which includes all involved grid points per stencil. For example, Fig. 1 shows a 6th order, 25-point SC whose footprint is $13^2 = 169$ on a 2-dimensional lattice. Such stencil is widely used in high-order finite-difference calculations [20, 21].

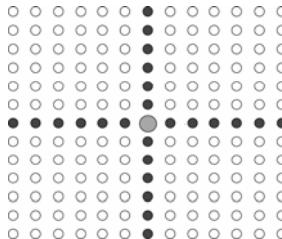


Fig. 1. 6-th order, 25-point SC whose footprint is 13^2 on a 2-dimensional lattice. The grid point of interest, \mathbf{r} , is shown as the central grey circle while the set of neighbor points, excluding \mathbf{r} , i.e., $\{\mathbf{r}' \mid \mathbf{r}' \in \text{neighbor}(\mathbf{r})\} - \{\mathbf{r}\}$, is illustrated as solid circles. White circles show the other lattice sites within the stencil footprint, which are not used for calculation of $v(\mathbf{r})$.

2.2 A High-Order Stencil Application for Seismic Wave Propagation

This subsection introduces our experimental application that simulates seismic wave propagation by employing a 3D equivalent of the stencil in Fig. 1 to compute spatial derivatives on uniform grids using a finite difference method. The 3D stencil kernel is highly off-diagonal (6-th order) and involves 37 points (footprint is $13^3 = 2,197$), i.e., each grid point interacts with 12 other grid points in each of the x, y and z Cartesian directions.

A typical problem space of the program includes several hundreds of grid nodes in each dimension amounting to an aggregate amount of millions of grid points. Our largest experiment has $400^3 = 64$ million points. For each grid point, the code allocates 5 floats to hold temporary arrays and intermediate physical quantities and the result, therefore its memory footprint is $5 \times 4\text{bytes} \times 400^3 = 1.28\text{GB}$. Thus, the application not only involves heavy computational requirements but also needs three orders of magnitude more memory in comparison with the cache size offered by multicore architectures in the current HPC literature.

3 Multilevel Parallelization Framework

This section discusses inter-node and intra-node optimizations we use and outlines our parallelization framework that combines: (1) inter-node parallelism by spatial decomposition; (2) intra-chip parallelism through multithreading; and (3) data-level parallelism via SIMD techniques.

3.1 Spatial Decomposition Based on Message Passing

Our parallel implementation essentially retains the computational methodology of the original sequential code. All of the existing subroutines and the data structures are retained. However, instead of partitioning the computational domain, we divide the 3D data space to a mutually exclusive and collectively exhaustive set of subdomains and distribute the data over the cluster assigning a smaller version of the same problem to each processor in the network, then employ an owner-computes rule.

Subdomains should have the same number of grid points to balance the computational load evenly. However, a typical stencil computation not only uses the grid points owned by the owner processor, but also requires boundary grid points to be exchanged among processors through a message passing framework. Therefore each subdomain is augmented with a surrounding buffer layer used for data transfer.

In a parallel processing environment, it is vital to understand the time complexity of communication and computation dictated by the underlying algorithm, to model performance of parallelism. For a d -dimensional finite difference stencil problem of order m and global grid size n run on p processors, the communication complexity is $O(m(n/p)^{(d-1)/d})$, whereas computation associated by the subdomain is proportional to the number of owned grid points, therefore its complexity is $O(mn/p)$ (which is linear in m , assuming a stencil geometrically similar to that shown in Fig.1). For a 3D problem, they reduce to $O(m(n/p)^{2/3})$ and $O(mn/p)$, which are proportional to the surface area and volume of the underlying subdomain, respectively. Accordingly, subdomains are selected to be the optimal orthorhombic box in the 3D domain minimizing the surface-to-volume ratio, $O((p/n)^{1/3})$, for a given number of processors. This corresponds to a cubic subvolume if processor count is cube of an integer number; otherwise a rectangular prism shaped subset is constructed by given logical network topology. Fig. 2(a) shows the decomposition of problem domain into cubical subvolumes, which are assigned to different processors.

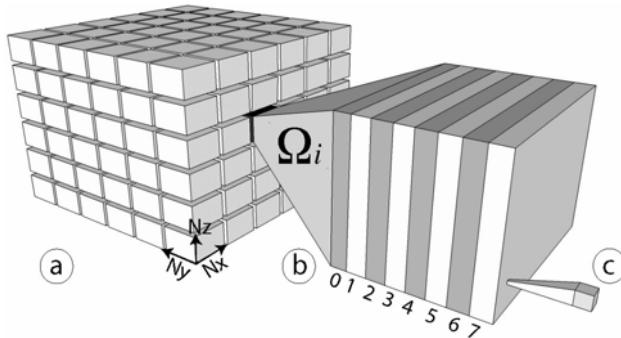


Fig. 2. Schematic of multilevel parallelization. In the uppermost level of multilevel parallelization, the total simulation space Ω is decomposed into several spatial sub-domains Ω_i of size $N_x \times N_y \times N_z$ points in corresponding directions where $\Omega = \cup \Omega_i$ and each domain is mapped onto a node in the cluster (a). Intra-node parallelization includes POSIX threading to exploit all available cores in the underlying multicore processor (b). It shows 8 threads and numerals as their IDs, which is intended for dual quad-core architecture nodes. Data-level parallelism constitutes the last level in the hierarchical parallelization methodology which we achieve through vectorization and implement by Streaming SIMD Extensions (SSE3) at Intel based architectures (c).

3.2 Intra-node Parallelism Based on Multithreading

On a cluster of Intel Clovertown, the code is implemented based on hierarchical spatial decomposition: (1) inter-node parallelization with the upper-level spatial decomposition into domains based on message passing; and (2) intra-node parallelization with the lower-level spatial decomposition within each domain through multithreading.

Fig. 3 shows the pseudocode for the threaded code section. There are 3 subroutines implementing this pseudocode in the actual application to perform computations in each of 3D Cartesian directions (represented by `toCompute` in the pseudocode) at each simulation step. 1D data array (`current` in Fig. 3) for grid points allocates x data to be unit stride direction, and y data to be N_x stride, finally z data with $N_x \times N_y$ stride. Threads spawned in `toCompute` direction evenly partition the domain in `lowStride` dimension and apply stencil formula for the partial spatial derivation with respect to the corresponding Cartesian coordinate, i.e., `toCompute`. Therefore, for a thread spawned to perform calculations in the z direction, `toCompute` is the z direction whereas `lowStride` corresponds to the x direction since y stride is larger than the unit stride for x. Before the actual computation is performed, consecutive points in `toCompute` direction, which are stride away in the `current` array, are packed into a temporary array, named `tempCurrent` in Fig. 3, to reduce effective access time to the same data throughout the computation. (This only applies to the implementations in non-unit access directions, y and z, as x data are already consecutive in `current`.) Finally, `tempCurrent` is traversed to compute new field values to update `next` array. Threads of the same direction can independently execute and perform updates on their assigned subdomain without introducing any locks until finally they are joined before the stencil for the other directions are calculated.

```

FOR each gridPoint in  $\Omega_i$  in highStride direction
  FOR each gridPoint in  $\Omega_i$  in lowStride direction
    SET packedGrid to 0
    FOR each gridPoint in  $\Omega_i$  in toCompute direction
      STORE current[gridPoint] in
      tempCurrent[packedGrid]
      INCREMENT packedGrid
    END FOR
    FOR each packedGridPoint in tempCurrent
      COMPUTE next[gridPoint] as the accumulation of
      packedGridPoint contributions
    END FOR
  END FOR
END FOR

```

Fig. 3. Pseudocode for the threaded code section. Since threads store their data in the `next` array, they avoid possible race condition and the program can exploit thread-level parallelism (TLP) in Intel Clovertown architecture.

3.3 Single Instruction Multiple Data Parallelism

Most modern computing platforms have incorporated single-instruction multiple-data (SIMD) extensions into their processors to exploit the natural parallelism of applications if the data can be SIMDized (i.e., if a vector instruction can simultaneously operate on a sequence of data items.). On Intel quadcore architectures, 128-bit wide registers can hold four single precision (SP) floating point (FP) numbers that are operated concurrently, and thus throughput is 4 SP FP operations per cycle and the ideal speedup is 4. However, determining how instructions depend on each other is critical to determine how much parallelism exists in a program and how that parallelism can be exploited. In particular, for finite-difference computations, instructions operating on different grid points are independent and can be executed simultaneously in a pipeline without causing any stalls. Because of this property, we can expose a finer grained parallelism through vectorization on top of our shared-memory Pthreads programming and manually implement using Streaming SIMD Extensions (SSE3) intrinsics on Intel based architectures, whose prototypes are given in Intel's `xmmmintrin.h` and `pmmmintrin.h`. In our SIMDization scheme, we first load successive neighbor grid points and corresponding stencil coefficients to registers using `_mm_load_ps`. Secondly, we perform computations using arithmetic SSE3 intrinsics (e.g. `_mm_add_ps`, `_mm_mul_ps`) and reduce the results by horizontally adding the adjacent vector elements (`_mm_hadd_ps`). Finally, we store the result back to the data array (`_mm_store_ss`). Our results have exhibited sub-optimal performance gain for the stencil orders not divisible by 4, as unused words in 4 word registers were padded with zeros, albeit SIMDization indeed contributed in intra-node speedup combined with multithreading for such stencil orders as well. More detailed performance-measurement results are outlined in Sections 4.3 and 4.4.

4 Performance Tests

The scalability and parallel efficiency of the seismic stencil code have been tested on various high-end computing platforms including dual-socket Intel Harpertown and Clovertown, AMD Barcelona based clusters at the high performance computing communications facility of University of Southern California (HPCC-USC). We also present weak-scaling benchmark results at 65,536 IBM BlueGene/P processors at the Argonne National Laboratory. The following subsections provide the details of our benchmarks and present the results.

4.1 Weak-Scaling

We have implemented the spatial decomposition using the message passing interface (MPI) [22] standard and have tested the inter-node scalability on IBM BlueGene/P and an HPCC-USC cluster. Fig. 4(a) shows the running and communication times of the seismic code at HPCC-USC, where each node has Intel dual quad-core Xeon E5345 (Clovertown) processors clocked at 2.33 GHz, featuring 4 MB L2 cache per die, together with 12 GB memory and are connected via 10-gigabit Myrinet. Here, we scale the number of grid points linearly with the number of processors: $10^6 p$ grid points on p processors. The wall-clock time per time step approximately stays constant as p increases. The weak-scaling parallel efficiency, the running time on 1 processor divided by that on p processors, is observed to be 0.992 on the Clovertown based cluster for our largest run on 256 processors. Fig. 4(b) also shows good weak-scaling parallel efficiency with increased grain size (8×10^6) on 65,536 BlueGene/P processors, where each node has 2 GB DDR2 DRAM and four 450 POWER PC processors clocked at 850 MHz, featuring a 32 KB instruction and data cache, a 2 KB L2 cache and a shared 8 MB L3 cache. Spatial decomposition is thus highly effective in terms of inter-node scalability of stencil computations up to 65,536 processors.

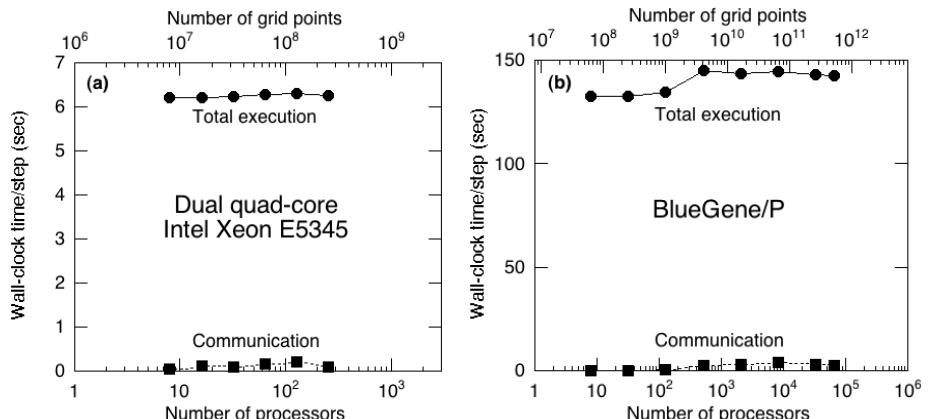


Fig. 4. Weak-scaling performance of spatial decomposition methodology, where the problem size scales: (a) smaller granularity (20 MB/process) on a dual quadcore Intel Xeon E5345 cluster; (b) larger granularity (160 MB/process) on BlueGene/P

4.2 Strong-Scaling

Strong-scalability (i.e., achieving large speedup for a fixed problem size on a large number of processors) is desirable for many finite-difference applications, especially for simulating long-time behaviors. Here, strong-scaling speedup is defined as the ratio of the time to solve a problem on one processor to that on p processors. Fig. 5 shows strong-scaling speedups for a fixed global problem size of 400^3 on several AMD and Intel quadcore based clusters. We observe superlinear speedups for increasing processor count on both AMD and Intel architectures, which may be interpreted as a consequence of the increasing aggregate cache size as explained below.

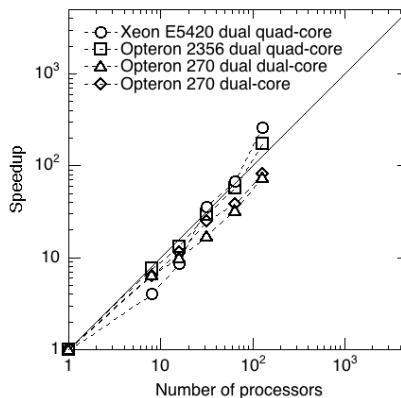


Fig. 5. Strong-scaling benchmark on various multicore CPUs. The speedup over single-processor run time is plotted as a function of the number of processors p . Intel Xeon E5420 shows superlinear speedup for smaller p as the aggregate cache size increases faster because of its 12 MB L2 cache. The solid line shows ideal speedup.

The original seismic stencil code suffers from high effective memory access time when the entire problem space is put into memory of one processor. We have analyzed cache and TLB misses for the original code by using Intel's Vtune Performance Analyzer and found that, for high-stride access computation that implements the partial spatial derivative in the z direction, the duration of page-walks amounts to more than 25% of core cycles throughout the thread execution, implying a high TLB miss rate resulting in greater effective memory access time. As we divide the problem space, the effective access time reduces since the hit ratio is higher for smaller problem size, even though the latency to the cache remains the same. It should be noted that, in our strong-scaling experiment, not only the processor count is increasing but also the size of aggregate caches from processors. With larger aggregate cache size, more (or even entire, depending on processor count) data can fit into the caches. The cache effect is most pronounced on Intel's Xeon E5420 (Harpertown) in Fig. 5. Intel's second-generation quadcore processor Harpertown features a shared 6 MB L2 cache per chip that accommodates two cores. This amounts to 12 MB of cache per multi-chip module (MCM), 6 times more than 2 MB

(4×512KB) L2 cache offered by AMD Opteron 2356 (Barcelona). As a result, Harpertown crosses over to the superlinear-scaling regime on a smaller number of processors compared with that for Barcelona (see Fig. 5).

4.3 Intra-node Optimization Results

In addition to the massive inter-node scalability demonstrated above, our framework involves the lower levels of optimization: First, we use multithreading explained in Section 3.2, implemented with Pthreads. Next, we vectorize both STORE and COMPUTE statements inside the triply nested for loops in Fig. 3 by using SSE3 intrinsics on the dual Intel Clovertown platform. We use Intel C compiler (icc) version 9.1 for our benchmarks.

Fig. 6(a) shows the reduction in clock time spent per simulation step due to multithreading and SIMDization optimizations. Corresponding speedups are shown in Fig. 6(b). To delineate the performances of multithreading and SIMDization approaches, we define a performance metric as follows. We use the clock time for one simulation step of single threaded, non-vectorized code to be the sequential run time T_s . We denote the parallel run time, $T_p(NU M_T H R E A D S)$, to be the clock time required for execution of one time step of the algorithm as a function of spawned thread number, in presence of both multithreading and SIMD optimizations. Then combined speedup, S_c , shown in Fig. 6(b) is equal to the ratio T_s/T_p as a function of thread number. We remove SIMD optimizations to quantify the effect of multithreading only, and measure the parallel running times for a variety of thread numbers, and state the multithreading speedup, S_t , with respect to T_s . Finally, we attribute the excess speedup, S_c/S_t , to SIMD optimizations.

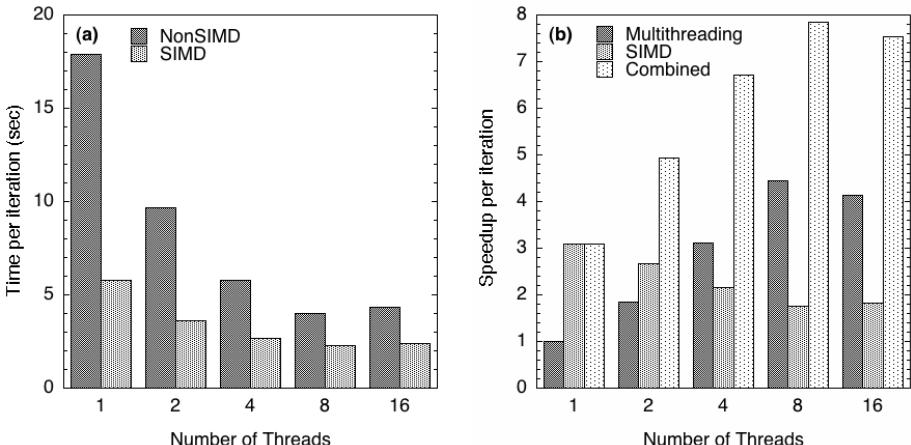


Fig. 6. The wall-clock time per iteration for non SIMD and SIMDized codes (a) and breakdown of speedups (due to multithreading and data-level parallelism along with the combined intra-node speedup) (b) as a function of the number of threads on dual quad-core Intel Clovertown. The best observed intra-node speedup is 7.85 with 8 threads spawned on 8 cores.

Fig. 6(b) shows the best speedup of 7.85 for 8 threads on a dual quadcore Intel Clovertown node. This amounts to $7.85/8 = 0.98$ intra-node parallel efficiency, even though separate contributions of SIMDization and multithreading are less than ideal. Multithreading speedup increases until thread number is equal to the available cores, after which we see performance degradation in the SIMD speedup due to more frequent exchanges of data between vector registers and main memory in case of greater number of threads.

4.4 Effects of Stencil Order

We have also studied the effect of stencil order on intra-node parallel efficiency. As mentioned in Section 3.3, our SIMDization scheme is suboptimal for our seismic stencil code, for which the stencil order is not a multiple of 4. In fact, it is 6 and uses two quad-word registers by padding one of them with 2 zeros. To reduce this overhead, we have increased the precision of our simulation and performed a similar computation with 7th order stencil (1 zero padding at one of the registers) and 8th order stencil (utilizing all 4 words of both of the 128-bit registers).

Fig. 7(a) shows the dependency of the combined multithreading+SIMD speedup, as defined in previous subsection, on stencil sizes. When all SIMD vectors are saturated, i.e., fully used, the best combined speedup is observed. In our implementation, SIMD vectors are saturated for 8th order stencil, and best speedup is observed with 8 threads on dual quadcore Intel Clovertown. In fact, we have achieved superlinear intra-node speedup of 8.7 with 8 threads on 8 cores. (Since we use in-core optimizations as well, it is not surprising to see more than 8-fold speedup.)

Fig. 7(b) shows the data parallelism contribution in the combined speedup. It normalizes the combined speedup in Fig. 7(a) with multithreaded, but non-vectorized speedup in order to quantify the speedup due to data level parallelism only. Saturating

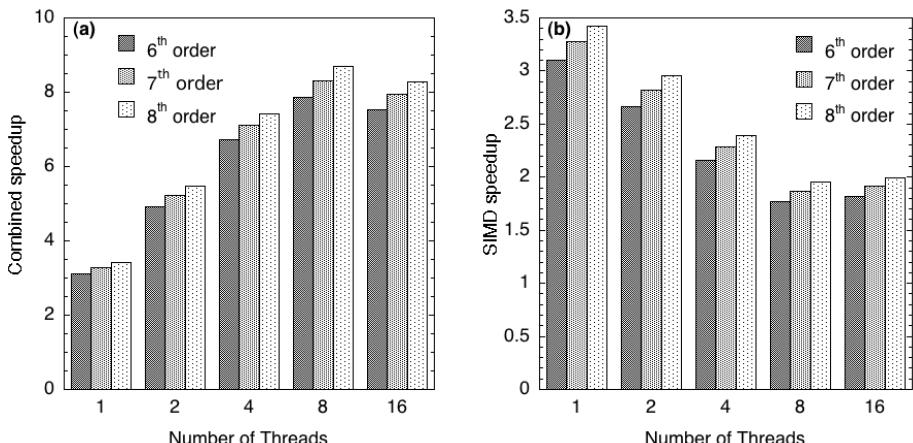


Fig. 7. Speedups for higher order stencils, which are required for higher precision in finite-differences calculation. Combined multithreading+SIMD (a) and SIMD (b) speedups are shown as a function of the number of threads for 6th, 7th and 8th order stencils.

vectors with actual data is observed to yield the highest performance gain, which corresponds to 8th order stencil in our implementation. We have observed 3.42-fold speedup due to data-level parallelism in 1 thread run for 8th order stencil, pointing out the highest SIMD efficiency of $3.42/4 = 0.85$ in Fig. 7(b). The figure also confirms the decreasing performance of SIMD as a function of thread count, as shown in Fig. 6(b).

5 Conclusions

In summary, we have developed a multilevel parallelization framework for high order stencil computations. We have applied our approach to a 6th-order stencil based seismic wave propagation code on a suite of multicore architectures. We have achieved superlinear strong-scaling speedup on Intel Harpertown and AMD Barcelona based clusters through the uppermost level of our hierarchical approach, i.e., spatial decomposition based on message passing. Our implementation has also attained 0.92 weak-scaling parallel efficiency at 65,536 BlueGene/P processors. Our intra-node optimizations included multithreading and data-level parallelism via SIMD techniques. Multithreading+SIMD optimizations achieved 7.85-fold speedup and 0.98 intra-node parallel efficiency on dual quadcore Intel Clovertown platform. We have quantified the dependency of data-level parallel efficiency on the stencil order, and have achieved 8.7-fold speedup for an extended problem employing 8th order stencil. Future work will address the analysis and improvement of flops performance, and perform lower-level optimizations at a broader set of emerging architectures.

Acknowledgments. This work was partially supported by Chevron—CiSoft, NSF, DOE, ARO, and DTRA. Scalability and performance tests were carried out using High Performance Computing and Communications cluster of the University of Southern California and the BlueGene/P at the Argonne National Laboratory. We thank the staff of the Argonne Leadership Computing Facility for their help on the BlueGene/P benchmark.

References

1. Dongarra, J., Gannon, D., Fox, G., Kennedy, K.: The impact of multicore on computational science software. *CTWatch Quarterly* 3, 11–17 (2007)
2. Barker, K.J., Davis, K., Hoisie, A., Kerbyson, D.J., Lang, M., Pakin, S., Sancho, J.C.: Entering the petaflop era: the architecture and performance of Roadrunner. In: *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, Austin (2008)
3. Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Pishker, W.L., Shalf, J., Williams, S.W., Yelick, K.A.: The landscape of parallel computing research: a view from Berkeley. University of California, Berkeley (2006)
4. Pakin, S.: Receiver-initiated message passing over RDMA networks. In: *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium*, Miami, Florida (2008)
5. Williams, S., Carter, J., Oliker, L., Shalf, J., Yelick, K.: Lattice Boltzmann simulation optimization on leading multicore platforms. In: *Proceedings of the 22nd International Parallel and Distributed Processing Symposium*, Miami, Florida (2008)

6. Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., Yelick, K.: Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In: Proceedings of the 2008 ACM/IEEE conference on Supercomputing. IEEE Press, Austin (2008)
7. Rivera, G., Tseng, C.-W.: Tiling optimizations for 3D scientific computations. In: Proceedings of the 2000 ACM/IEEE conference on Supercomputing. IEEE Computer Society Press, Dallas (2000)
8. Frigo, M., Strumpen, V.: Cache oblivious stencil computations. In: Proceedings of the 2005 ACM/IEEE conference on Supercomputing. ACM, Cambridge (2005)
9. Wonnacott, D.: Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In: Proceedings of the 14th IEEE International Parallel and Distributed Processing Symposium, Cancun, Mexico (2000)
10. Renganarayanan, L., Harthikote-Matha, M., Dewri, R., Rajopadhye, S.V.: Towards optimal multi-level tiling for stencil computations. In: Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium, Long Beach, California (2007)
11. PARKBENCH: PARallel Kernels and BENCHmarks,
<http://www.netlib.org/parkbench>
12. Kamil, S., Datta, K., Williams, S., Oliker, L., Shalf, J., Yelick, K.: Implicit and explicit optimizations for stencil computations. In: Proceedings of the 2006 Workshop on Memory System Performance and Correctness. ACM, San Jose (2006)
13. Schreiber, R., Dongarra, J.: Automatic blocking of nested loops. Technical Report, University of Tennessee (1990)
14. Desprez, F., Dongarra, J., Rastello, F., Robert, Y.: Determining the idle time of a tiling: new results. *Journal of Information Science and Engineering* 14, 167–190 (1998)
15. Bromley, M., Heller, S., McNerney, T., Steele, J.G.L.: Fortran at ten gigaflops: the connection machine convolution compiler. In: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation. ACM, Toronto (1991)
16. Bleck, R., Rooth, C., Hu, D., Smith, L.T.: Salinity-driven Thermocline Transients in a Wind- and Thermohaline-forced Isopycnic Coordinate Model of the North Atlantic. *Journal of Physical Oceanography* 22(12), 1486–1505 (1992)
17. Harlick, R., Shapiro, L.: Computer and Robot Vision. Addison Wesley, Reading (1993)
18. Nakano, A., Vashishta, P., Kalia, R.K.: Multiresolution molecular dynamics algorithm for realistic materials modeling on parallel computers. *Computer Physics Communications* 83 (1994)
19. Taflove, A., Hagness, S.C.: Computational Electrodynamics: The Finite-Difference Time-Domain Method, 3rd edn. Artech House, Norwood (2005)
20. Shimojo, F., Kalia, R.K., Nakano, A., Vashishta, P.: Divide-and-conquer density functional theory on hierarchical real-space grids: Parallel implementation and applications. *Physical Review B* 77 (2008)
21. Stathopoulos, A., Öğüt, S., Saad, Y., Chelikowsky, J.R., Kim, H.: Parallel methods and tools for predicting material properties. *Computing in Science and Engineering* 2, 19–32 (2000)
22. Snir, M., Otto, S.: MPI-The Complete Reference: The MPI Core. MIT Press, Cambridge (1998)

Using OpenMP vs. Threading Building Blocks for Medical Imaging on Multi-cores

Philipp Kegel, Maraike Schellmann, and Sergei Gorlatch

University of Münster, Germany

{p.kegel,schellmann,gorlatch}@uni-muenster.de

Abstract. We compare two parallel programming approaches for multi-core systems: the well-known OpenMP and the recently introduced Threading Building Blocks (TBB) library by Intel®. The comparison is made using the parallelization of a real-world numerical algorithm for medical imaging. We develop several parallel implementations, and compare them w.r.t. programming effort, programming style and abstraction, and runtime performance. We show that TBB requires a considerable program re-design, whereas with OpenMP simple compiler directives are sufficient. While TBB appears to be less appropriate for parallelizing existing implementations, it fosters a good programming style and higher abstraction level for newly developed parallel programs. Our experimental measurements on a dual quad-core system demonstrate that OpenMP slightly outperforms TBB in our implementation.

1 Introduction

Modern CPUs, even on desktop computers, are increasingly employing multiple cores to satisfy the growing demand for computational power. Thus, easy-to-use parallel programming models are needed to exploit the full potential of parallelism. Early parallel programming models (e.g. Pthreads) usually allow for flexible parallel programming but rely on low-level techniques: The programmer has to deal explicitly with processor communication, threads and synchronization, which renders parallel programming tedious and error-prone. Several techniques exist to free programmers from low-level implementation details of parallel programming. One approach is to extend existing programming languages with operations to express parallelism. OpenMP is an example of this approach. Another approach is to introduce support for parallel programming within a library. Recently, Intel® published one such library, Threading Building Blocks (TBB), that adds support for high-level parallel programming techniques to C++.

The OpenMP application programming interface is quite popular for shared-memory parallel programming [3]. Basically, OpenMP is a set of compiler directives that extend C/C++ and Fortran compilers. These directives enable the user to explicitly define parallelism in terms of constructs for single program multiple data (SPMD), work-sharing and synchronization. OpenMP also provides some library functions for accessing the runtime environment. At compile

time, multi-threaded program code is generated based on the compiler directives. OpenMP has been designed to introduce parallelism in existing sequential programs. In particular, loop-based data parallelism can be easily exploited.

Threading Building Blocks (TBB) is a novel C++ library for parallel programming, which can be used with virtually every C++ compiler. In TBB, a program is described in terms of fine-grained tasks. Threads are completely hidden from the programmer. TBB's idea is to extend C++ with higher-level, task-based abstractions for parallel programming; it is not just a replacement for threads [6]. At runtime, TBB maps tasks to threads. Tasks are more light-weight than threads, because they cannot be preempted, such that no context switching is needed. They may be executed one after another by the same thread. This reduces scheduling overhead, which usually occurs when each program task is defined and performed by a single thread. Tasks also can be re-scheduled at runtime to load-balance threads. This mechanism is referred to as "task-stealing" [6]. TBB provides skeletons for common parallel programming patterns, like map, reduce or scan. Thereby it facilitates high-level parallel programming.

This paper studies the use of OpenMP and TBB for parallelizing an existing sequential implementation of an algorithm for image reconstruction. We compare parallel implementations of this algorithm based on OpenMP and TBB, using such criteria as programming effort, programming style and abstraction, and runtime performance. In Section 2, the algorithm that we want to parallelize and its field of application are presented. The actual parallelization is described in Section 3. Here, we provide some code examples and a detailed description of several parallel implementations of the algorithm based on OpenMP and TBB. In Section 4, we compare our TBB- and OpenMP-based implementations. We present runtime performance and discuss programming style. Finally, we conclude with an evaluation of TBB and OpenMP in Section 5 with respect to our algorithm and describe our ongoing and future work.

2 Positron Emission Tomography (PET) Imaging

In Positron Emission Tomography (PET), a radioactive substance is injected into a specimen (mice or rats in our case). Afterwards, the specimen is placed within a scanner, a device that is facilitated with several arrays of sensors. As the particles of the applied substance decay, two positrons are emitted (hence the name PET) in opposite directions. The "decay events" are detected by the sensors, usually by two opposite sensors in parallel. The scanner records these events, with each record comprising a timestamp and the positions of the two sensors.

ListMode Ordered Subset Expectation Maximization [5] (LM OSEM) is a block-iterative algorithm for 3D image reconstruction. LM OSEM takes a set of the aforementioned recorded events and splits them into s equally sized subsets. For each subset $l \in 0, \dots, s - 1$ the following computation is performed:

$$f_{l+1} = f_l c_l; \quad c_l = \frac{1}{A_N^t \mathbf{1}} \sum_{i \in S_l} (A_i)^t \frac{1}{A_i f_l}. \quad (1)$$

Here $f \in \mathbb{R}^n$ is a 3D image in vector form with dimensions $n = (X \times Y \times Z)$. $A \in \mathbb{R}^{m \times n}$, element a_{ik} of row A_i is the length of intersection of the line between the two detectors of event i with voxel k of the reconstruction region, computed using Siddon's algorithm [9]. Each subset's computation takes its predecessor's output image as input and produces a new, more precise image.

The overall structure of a sequential LM OSEM implementation is shown in Listing 1. It comprises three nested loops, one outer loop with two inner loops. The outer loop iterates over the subsets. The first inner loop iterates over a subset to compute the summation part of c_l . The second inner loop iterates over all elements of f_l and c_l to compute f_{l+1} .

```

for (int l = 0; l < subsets; l++) {
    /* read subset */

    /* compute c_l */
    #pragma omp parallel
    {
        #pragma omp for schedule(static)
        for (int i = 0; i < subset_size; i++) {
            ...
        }
    } /* end of parallel region */

    /* compute f_l+1 */
    #pragma omp parallel for schedule(static)
    for (int k = 0; k < image_size; k++) {
        if (sens[k] > 0.0 && c_l[k] > 0.0)
            f[k] = f[k] * c_l[k] / sens[k];    }  }

```

Listing 1. The original sequential implementation comprises one outer loop with two nested inner loops. For parallelization it is augmented with OpenMP compiler directives.

A typical 3D image reconstruction processing 6×10^7 million input events for a $150 \times 150 \times 280$ PET image takes more than two hours on a common PC. Several parallel implementations for systems with shared- and distributed-memory, as well as hybrid systems have been developed to reduce the algorithm's runtime [4,7]. Also an implementation for Compute Unified Device Architecture (CUDA) capable graphics processing units is available [8].

3 Parallelizing the LM OSEM Algorithm

Because of the data dependency between the subset's computations (implied by $f_{l+1} = f_l c_l$), the subsets cannot be processed in parallel. But the summation part of c_l and the computation of f_{l+1} are well parallelizable. (Additional parts of the implementation, e.g., for performing an image convolution to improve image

quality, can also be parallelized. However, none of these program parts have to do directly with LM OSEM and will be omitted in the sequel.)

3.1 Loop Parallelization

To parallelize the summation part of c_l and the computation of f_{l+1} , we have to parallelize the two inner loops of the LM OSEM algorithm. OpenMP and TBB both offer constructs to define this kind of data parallelism.

In OpenMP, two constructs are used to parallelize a loop: the *parallel* and the *loop* construct. The parallel construct, introduced by a `parallel` directive, declares the following code section (the *parallel region*) to be executed in parallel by a team of threads. Additionally, the loop construct, introduced by a `for` directive, is placed within the parallel region to distribute the loop's iterations to the threads executing the parallel region. We parallelize the inner loop using these two constructs, as shown in Listing 1. For the first inner loop, each thread must perform some preliminary initializations. The according statements are placed at the parallel region's beginning. The second inner loop does not need any initializations. Therefore, we use the `parallel for` directive, which is a shortcut declaration of a parallel construct with just one nested loop construct. Apart from the additional compiler directives, no considerable changes were made to the sequential program. Thus, an OpenMP-based parallel implementation of LM OSEM is easily derived from a sequential implementation of the LM OSEM algorithm written in C 7, see Listing 1.

OpenMP supports several strategies for distributing loop iterations to threads. The strategy is specified via the `schedule` clause, which is appended to the `for` directive. In our application, we expect that the workload is evenly distributed in iteration ranges of reasonable size (greater than 10000 events). Therefore, a static and even distribution of iterations to threads will be sufficient for our application. OpenMP's `static` scheduling strategy suits our needs. If not specified otherwise, it creates evenly sized blocks of loop iterations for all threads of a parallel region. Thereby, the block size implicitly is defined as n/p , where n is the number of iterations and p is the number of threads.

In TBB, parallel loops are defined using the `parallel_for` template, which is applied in two steps as proposed by Reinders 6:

1. A class is created, which members correspond to the variables that are outside the scope of the loop to be parallelized. The class has to overload the `()` operator method so that it takes an argument of type `blocked_range<T>` that defines a chunk of the loop's iteration space which the operator's caller should iterate over. The method's body takes the original code for the loop. Then, the arguments for the loop's iteration bounds are replaced by calls to methods that return the iteration space's beginning and end.
2. The code of the original loop is replaced by a call to the `parallel_for` pattern. The pattern is called with two arguments, the loop's iteration space and an instance of the previously defined class, called the *body object*. The iteration space is defined by an instance of the `blocked_range<T>` template class. It takes the iteration space's beginning and end and a *grain size* value.

A parallel implementation of LM OSEM using TBB thus requires to re-implement the two inner loops. The most obvious difference to OpenMP is the use of C++ instead of C. TBB's `parallel_for` construct is a C++ template and takes a C++ class as parameter. Actually, this technique is a workaround for C++'s missing support of *lambda expressions*. With lambda expression, blocks of code can be passed as parameters. Thus, the code of the body object could be passed to the `parallel_for` template *in-place* without the overhead of a separate class definition [6].

According to step 1, we create a class for each of the loops. Within these classes, we change the loops' iteration bounds and provide member variables that save the original outer context of the loops (see Listing 2). We determined these variables by carefully analyzing our existing implementation of LM OSEM.

```
class ImageUpdate {
    double *const f, *const c_l;
    double *const sens,
    public:
        ImageUpdate(double *f, double *sens, double *c_l) :
            f(f), sens(sens), c_l(c_l) {}

        void operator() (const blocked_range<int>& r) const {
            for (int k = r.begin(); k != r.end(); k++) {
                if (sens[k] > 0.0 && c_l[k] > 0.0)
                    f[k] *= c_l[k] / sens[k]; }
        }
};
```

Listing 2. The loop to be parallelized is embedded into a separate class that defines a method `operator()`

Then, following step 2, we are able to replace the loops by calls to TBB's `parallel_for` template (see Listing 3).

```
for (int l = 0; l < subsets; l++) {
    /* read subset */

    /* compute c_l */
    parallel_for(
        blocked_range<int>(0, subset_size, GRAIN_SIZE),
        SubsetComputation(f, c_l, event_buffer, precision));

    /* compute f_l+1 */
    parallel_for(
        blocked_range<int>(0, image_size, GRAIN_SIZE),
        ImageUpdate(f, sens, c_l)); }
```

Listing 3. A call to the `parallel_for` template replaces the loop

The grain size value specifies a reasonable maximum number of iterations that should be performed by a single processor. Unlike OpenMP, which provides multiple scheduling strategies, TBB always uses a fixed scheduling strategy that is affected by the selected grain size. If the size of an iteration range is greater than the specified grain size, the `parallel_for` template recursively splits it into disjoint iteration ranges. The grain size value thus serves as a lower bound for the tasks' size. If each task comprised a single iteration, this would result in a massive task scheduling overhead. Internally, the `parallel_for` template creates a task for each iteration range in a divide and conquer manner. Finally, all tasks are processed independently; i.e., in parallel.

TBB's grain size value is comparable to an optional *block size* value, which can be specified for OpenMP's `schedule` clause. Both specify a threshold value for chunks of iterations, to avoid too large or too small work pieces for each thread. However, the semantics of the block size value slightly differ from the grain size value and depend on the selected scheduling strategy. Reinders [6] proposes to select a grain size, such that each iteration range (task) takes at least 10,000 to 100,000 instructions to execute. According to this rule of thumb we select a grain size value of 1,000 for the first inner loop and a value of 10,000 for the second inner loop.

3.2 Thread Coordination

Within the first inner loop (summation part of c_l) all threads perform multiple additions to arbitrary voxels of a common intermediate image. Hence, possible race conditions have to be prevented. There are two basic techniques for this:

1. Mutexes: The summation part is declared mutually exclusive, such that only one thread at a time is able to work on the image.
2. Atomic operations: The summation is performed as an atomic operation.

In OpenMP, both techniques are declared by appropriate directives. Mutexes are declared by using the *critical* construct. Similarly to the aforementioned parallel construct, it specifies a mutual exclusion for the successive code section (see Listing 4).

```
/* compute c_l */
...
#pragma omp critical
while (path_elements[m].coord != -1) {
    c_l[path_elem[m].coord] += c * path_elem[m].length
} /* end of critical section */
```

Listing 4. Mutex in OpenMP: The *critical* construct specifies a mutex for the successive code section (critical region)

OpenMP's *atomic* construct is used similarly to the *critical* construct (see Listing 5). It ensures that a specific storage location is updated without interruption (atomically) [1]. Effectively, this ensures that the operation is executed

only by one thread at a time. However, while a critical region semantically locks the whole image, an atomic operation just locks the voxel it wants to update. Thus atomic operations in our case may be regarded as fine-grained locks.

```
/* compute c_l */
...
while (path_elements[m].coord != -1) {
    path_elem[m].length *= c;
    #pragma omp atomic
    c_l[path_elem[m].coord] += path_elem[m].length; }
```

Listing 5. Atomic operation in OpenMP: The *atomic* construct ensures that the voxels of the intermediate image are updated atomically

Besides, mutual exclusion can be implemented explicitly by using low-level library routines. With these routines, in general, a *lock variable* is created. Afterwards a lock for this variable is acquired and released explicitly. This provides a greater flexibility than using the *critical* or *atomic* construct.

TBB provides several mutex implementations that differ in properties like scalability, fairness, being re-entrant, or how threads are prevented from entering a critical section [6]. *Scalable* mutexes do not perform worse than a serial execution, even under heavy contention. Fairness prevents threads from starvation and, for short waits, *spinning* is faster than sending waiting threads to sleep. The simplest way of using a mutex in TBB is shown in Listing 6. The mutex lock variable is created explicitly, while a lock on the mutex is acquired and released implicitly. Alternatively, a lock can be acquired and released explicitly by appropriate method calls, similarly to using the OpenMP library functions.

```
tbb::mutex mtx; /* create mutex */
...
{ /* beginning of lock scope */
    tbb::mutex::scoped_lock lock(mtx); /* acquire lock on mutex */
    while (path_elements[m].coord != -1) {
        c_l[path_elem[m].coord] += c * path_elem[m].length; }
} /* end of scope, release lock */
```

Listing 6. Mutex in TBB: The constructor of `scoped_lock` implicitly acquires a lock, while the destructor releases it. The brackets keep the lock's scope as small as possible.

Atomic operations can only be performed on a template data type that provides a set of methods (e.g. `fetchAndAdd`) for these operations. Thus, the original data type (e.g., the image data type of LM OSEM) would have to be changed throughout the whole program. In our application, the voxels of an image are represented by double-precision floating-point numbers. However, TBB only supports atomic operations on integral types [6]. Therefore, with TBB, we cannot use atomic operations in our implementation of the LM OSEM algorithm.

4 Comparison: OpenMP vs. TBB

To compare OpenMP and TBB concerning our criteria of interest, we create several parallel implementations of LM OSEM using OpenMP and TBB. Two versions of an OpenMP-based implementation are created using the *critical* and the *atomic* construct, respectively. Moreover, we create three versions of our implementation based on TBB using TBB's three basic mutex implementations: a `spin_mutex` (not scalable, not fair, spinning), a `queuing_mutex` (scalable, fair, spinning) and a wrapper around a set of operation system calls (just called `mutex`) that provides mutual exclusion (scalability and fairness OS-dependent, not spinning).

4.1 Programming Effort, Style, and Abstraction

The previous code examples clearly show that TBB requires a thorough redesign of our program, even for a relatively simple pattern like `parallel_for`. Our TBB-based implementations differ greatly from the original version. We had to create additional classes, one for each loop to be parallelized, and replace the parallelized program parts by calls to TBB templates. Basically, this is because TBB uses library functions that depend on C++ features like object orientation and templates. Consequently, the former C program becomes a mixture of C and C++ code. The most delicate issue was identifying the variables that had to be included within the class definition of TBB's `parallel_for` body object.

Parallelizing the two inner loops of the original LM OSEM implementation using OpenMP is almost embarrassingly easy. Inserting a single line with compiler directives parallelizes a loop without any additional modifications. Another single line implements a mutual exclusion for a critical section or the atomic execution of an operation. Also, in contrast to TBB, we do not have to take any measures to change variable scopes. OpenMP takes care of most details of thread management.

Regarding the number of lines of code, OpenMP is far more concise than TBB. The parallel constructs are somewhat hidden in the program. For example, the end of the parallel region that embraces the first inner loop of LM OSEM is hard to identify in the program code. TBB, on the other hand, improves the program's structure. Parallel program parts are transferred into separate classes. Though this increases the number of lines of code, the main program becomes smaller and more expressive through the use of the `parallel_for` construct.

The parallel constructs of TBB and OpenMP offer a comparable level of abstraction. In neither case we have to work with threads directly. TBB's `parallel_for` template resembles the semantics of OpenMP's parallel loop construct. OpenMP's parallel loop construct can be configured by specifying a scheduling strategy and a block size value, whereas TBB relies solely on its task-scheduling mechanism. However, OpenMP's parallel loop construct is easier to use, because the `schedule` clause may be omitted.

Regarding thread coordination, OpenMP and TBB offer a similar set of low-level library routines for locking. However, OpenMP additionally provides

compiler directives which offer a more abstract locking method. Also, for our application, TBB has a serious drawback regarding atomic operations: these operations are only supported for integral types (and pointers).

4.2 Runtime Performance

To give an impression of the performance differences between OpenMP and TBB with respect to our application, we compare the OpenMP-based parallel implementations with the ones based on TBB.

We test our implementations on a dual quad-core (AMD OpteronTM 2352, 2.1 GHz) system. Each core owns a 2×64 KB L1 cache (instruction/data) and 512 KB L2 cache. On each processor, four cores share 2 MB L3 cache and 32 GB of main memory. The operating system is Scientific Linux 5.2.

We run the LM OSEM algorithm on a test file containing about 6×10^7 events retrieved from scanning a mouse. To limit the overall reconstruction time in our tests, we process only about 6×10^6 of these events. The events are divided into ten subsets of one million events each. Thus, we get subsets that are large enough to provide a reasonable intermediate image (c_l), while the number of subsets is sufficient for a simple reconstruction. Each program is re-executed ten times. Afterwards, we calculate the average runtime for processing a subset. We repeat this procedure using 1, 2, 4, 8, and 16 threads. The number of threads is specified by an environment variable in OpenMP or by a parameter of the task scheduler's constructor in TBB, respectively.

Figure 1 shows a comparison of the results of the implementations of LM OSEM using mutexes. Most of the implementations show a similar scaling behavior. Exceptions are the implementations based on TBB's `mutex` wrapper and

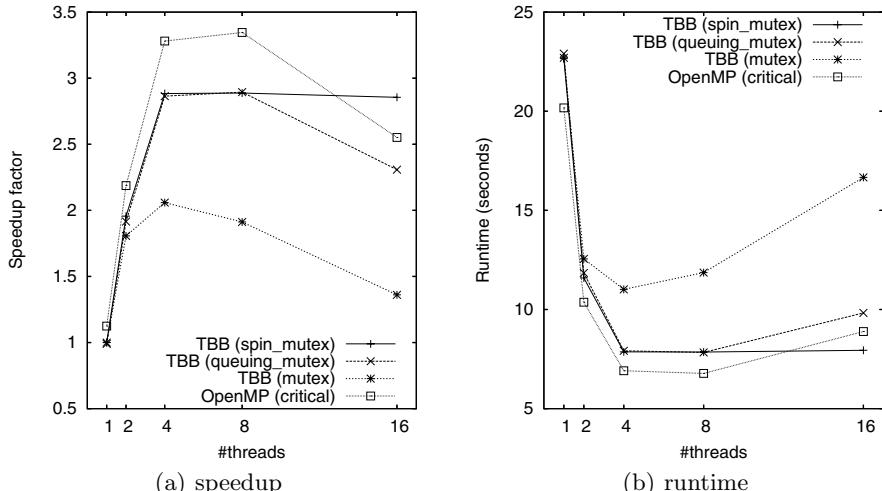


Fig. 1. Runtime and speedup for a single subset iteration of different implementations of LM OSEM

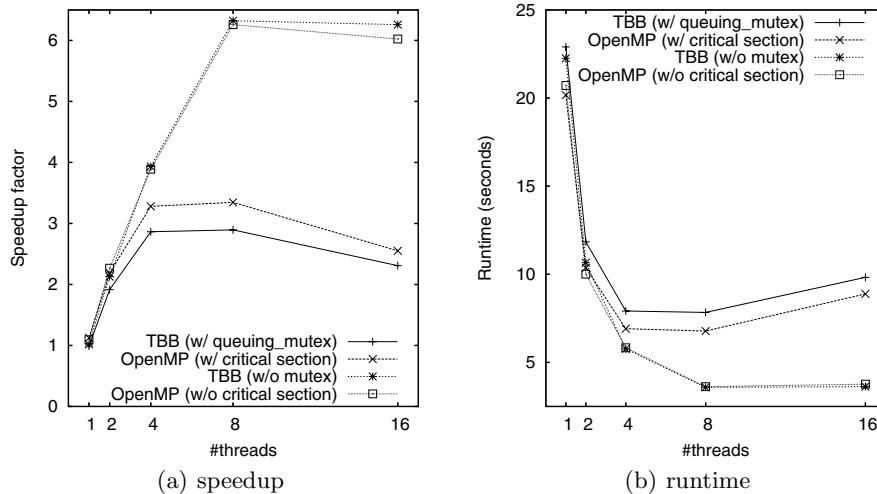


Fig. 2. Runtime and speedup for a single subset iteration of different implementations of LM OSEM. Without thread coordination both implementations perform equally.

spin_mutex. The implementation based on the `mutex` wrapper performs worst of all implementations. In particular, it shows a much worse scaling behavior. With 16 threads (oversubscription), the `spin_mutex`-based implementation performs best of all. However, OpenMP's *critical* construct outperforms most of TBB's mutex implementations.

To clarify the reasons for TBB's inferior performance as compared to OpenMP, we repeat our experiments with two implementations that do not prevent race

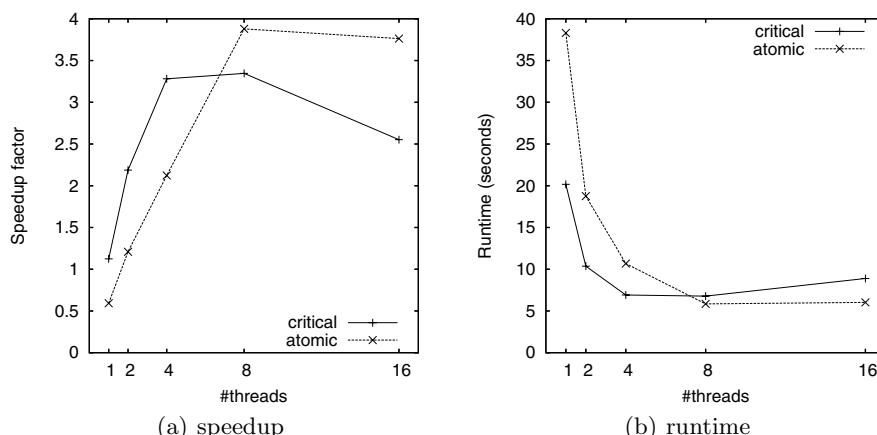


Fig. 3. Runtime and speedup for a single subset iteration of the OpenMP-based LM OSEM implementations

conditions in OpenMP or TBB, respectively. The results (see Figure 2) show that without a mutex (or critical section), TBB and OpenMP perform virtually equally well.

A comparison of the two OpenMP-based implementations of LM OSEM (see Figure 3) reveals a remarkable difference. The runtime of the single-threaded case shows that using atomic operations slows down our implementation by about 50%. However, when the number of threads increases, the implementation using atomic operations provides an almost linear speedup. With about 8 threads, it finally outperforms the OpenMP-based implementation using the *critical* construct and all implementations using TBB's mutexes.

One should keep in mind that the performance of an OpenMP program heavily depends on the used compiler (Intel C/C++ compiler version 10.1 in our case). While the TBB library implements parallelization without any special compiler support, with OpenMP the parallelization is done by the compiler. Hence, the presented results do not apply to every OpenMP-capable compiler in general.

5 Conclusion

Using OpenMP, we can parallelize LM OSEM with little or no program redesign. It can be easily used for parallelizing our existing sequential implementation. TBB, on the other hand, fosters a structured, object-oriented programming style, which comes along with the cost of re-writing parts of the program. Hence, TBB is more appropriate for the creation of a new parallel implementation of LM OSEM. Besides, TBB is C++ based. Hence, one might argue that using TBB also requires object-oriented programming. *Lambda expressions* (see Section 3.1) will probably become part of the next C++ standard. This might enable TBB to implement loop parallelization *in place*, such that at least the effort for loop parallelization might be reduced to a level comparable to that of OpenMP [6].

With TBB, we can implement the same kind of data-parallelism as with OpenMP. Also, both approaches provide an abstract programming model, which shields the programmer from details of thread programming.

Regarding runtime, we get the best results when using OpenMP. TBB offers a comparable scaling behavior, but does not achieve a similar absolute performance. Our experimental results show that TBB's inferior performance is probably caused by its non-optimal mutex implementations, while its `parallel_for` template provides equal performance. Besides, TBB lacks atomic operations for floating-point data types, while such operations provide the best performance when used in our OpenMP-based implementation.

In conclusion, OpenMP apparently is a better choice for easily parallelizing our sequential implementation of the LM OSEM algorithm than TBB. OpenMP offers a fairly abstract programming model and provides better runtime performance. Particularly thread synchronization, which is crucial in our application, is easier to implemented when using OpenMP.

Nevertheless, apart from the `parallel_for` template, TBB offers some additional constructs (e.g. *pipeline*), which provide a higher level of abstraction. In

particular, these constructs allow the implementation of task-parallel patterns. We excluded these constructs from the comparison, because OpenMP contains no comparable features. If we started a new implementation of the LM OSEM algorithm with parallelism in mind, we might much easier exploit parallelism in TBB than in OpenMP by using these high-level constructs. For example, we might concurrently fetch and process events and calculate c_l by using TBB's pipeline construct. Especially, we would not have to take care about thread synchronization explicitly as we have to do in our current implementations. OpenMP does not provide such a construct, so that it would have to be implemented from scratch. Currently, we are working on implementations of LM OSEM using these features to exploit a higher level of programming.

In our future experiments, we also plan to analyze the exact influence of grain and block size and scheduling strategy (see section 3.1) for different amounts of input data on the program performance.

The most important feature of TBB is its task-based programming model. Usually, templates shield the programmers from directly using tasks. The latest version 3.0 of OpenMP also includes a task-based parallel programming model [2]. We are going to study how far this task model is comparable to TBB and whether it increases OpenMP's performance.

References

1. OpenMP.org – The OpenMP API specification for parallel programming, <http://openmp.org/>
2. OpenMP Architecture Review Board. OpenMP Application Program Interface (May 2008)
3. Chapman, B., Jost, G., van der Pas, R.: Using OpenMP - Portable Shared Memory Parallel Programming. MIT Press, Cambridge (2007)
4. Hoefler, T., Schellmann, M., Gorlatch, S., Lumsdaine, A.: Communication optimization for medical image reconstruction algorithms. In: Lastovetsky, A., Kechadi, T., Dongarra, J. (eds.) EuroPVM/MPI 2008. LNCS, vol. 5205, pp. 75–83. Springer, Heidelberg (2008)
5. Reader, A.J., Erlandsson, K., Flower, M.A., Ott, R.J.: Fast accurate iterative reconstruction for low-statistics positron volume imaging. Physics in Medicine and Biology 43(4), 823–834 (1998)
6. Reinders, J.: Outfitting C++ for Multi-core Processor Parallelism - Intel Threading Building Blocks. O'Reilly, Sebastopol (2007)
7. Schellmann, M., Kösters, T., Gorlatch, S.: Parallelization and runtime prediction of the listmode osem algorithm for 3d pet reconstruction. In: IEEE Nuclear Science Symposium and Medical Imaging Conference Record, San Diego, pp. 2190–2195. IEEE Computer Society Press, Los Alamitos (2006)
8. Schellmann, M., Vörding, J., Gorlatch, S., Meiländer, D.: Cost-effective medical image reconstruction: from clusters to graphics processing units. In: CF 2008: Proceedings of the 2008 conference on Computing Frontiers, pp. 283–292. ACM, New York (2008)
9. Siddon, R.L.: Fast calculation of the exact radiological path for a three-dimensional CT array. Medical Physics 12(2), 252–255 (1985)

Parallel Skeletons for Variable-Length Lists in SkeTo Skeleton Library

Haruto Tanno and Hideya Iwasaki

The University of Electro-Communications
1-5-1 Chofugaoka, Chofu, Tokyo 182-8585 Japan
tanno@ipl.cs.uec.ac.jp, iwasaki@cs.uec.ac.jp

Abstract. Skeletal parallel programming is a promising solution to simplify parallel programming. The approach involves providing generic and recurring data structures like lists and parallel computation patterns as skeletons that conceal parallel behaviors. However, when we focus on lists, which are usually implemented as one-dimensional arrays, their length is restricted and fixed in existing data parallel skeleton libraries. Due to this restriction, many problems cannot be coded using parallel skeletons. To resolve this problem, this paper proposes parallel skeletons for lists of variable lengths and their implementation within a parallel skeleton library called SkeTo. The proposed skeletons enable us to solve a wide range of problems including those of twin primes, Knight's tour, and Mandelbrot set calculations with SkeTo. We tested and confirmed the efficiency of our implementation of variable-length lists through various experiments.

1 Introduction

Writing efficient parallel programs is difficult, because we have to appropriately describe synchronization, inter-process communications, and data distributions among processes. Many studies have been devoted to this problem to make parallel programming easier. Programming with parallel computation patterns (or skeletons), or skeletal parallel programming [8][10][15] in short, has been proposed as one promising solution. Skeletons abstract generic and recurring patterns within parallel programs and conceal parallel behaviors within their definitions. They are typically provided as a library. Skeletons can be classified into two groups: *task parallel* skeletons and *data parallel* skeletons. Task parallel skeletons capture the parallelism by the execution of several different tasks, while data parallel skeletons capture the simultaneous computations on the partitioned data among processors. Parallel skeleton libraries enable users to develop a parallel program by composing suitable skeletons as if it were a sequential program.

However, in data parallel skeletons offered by existing parallel skeleton libraries, the length of a list, which is usually implemented as an one-dimensional array, is fixed due to an implementation reason. Thus, operations that dynamically and destructively change a list's length without allocating a new list within a memory area, e.g., adding elements to a list or removing elements from it, are not permitted. As a result, some problems cannot be solved efficiently or cannot be easily or concisely described by using existing data parallel skeletons.

```

numbers = [2,3,4,5,6,...,50] ; // list of numbers
primes = [] ; // list of prime numbers
twin_primes = [] ; // list of twin primes
// make a list of primes using the Sieve of Eratosthenes
do {
    prime = take front element from numbers
    remove elements that are divisible by prime from numbers
    add prime to primes
} while ( prime <= sqrt(MAX) );
concatenating primes and numbers
// primes is [2,3,5,7,11,13,...,47]
// make a list of twin primes from a list of primes
twin_primes = making pair of adjacent prime numbers
// twin_primes is [(2,3),(3,5),(5,7),(7,11),(11,13),...,(43,47)]
remove each pair of prime numbers whose difference is not two
// twin_primes is [(3,5),(5,7),(11,13),(17,19),(29,31),(41,43)]

```

Fig. 1. Pseudo-code for computing twin primes

For example, consider a problem to compute twin primes. Twin primes are pairs of prime numbers that differ by two, e.g., (3, 5) and (5, 7). A pseudo-code for this problem that computes twin primes of less than or equal to fifty is shown in Fig. 1. First, we create a list of prime numbers from a list of integers by using the Sieve of Eratosthenes. Second, we create a list of pairs of adjacent prime numbers. Finally, we remove every pair whose difference is not two to obtain a list of twin primes. A list of fixed length does not offer operations such as removing elements from a list, adding elements to it, or concatenating two lists, since these operations obviously change the length of the list. If a list of variable length (a list whose length may dynamically change) were supported, we could solve this problem based on the concise code in Fig. 1.

We could emulate a variable-length list by using a fixed-length list whose elements were pairs of a Boolean value and an integer. The Boolean value in an element indicates whether the element is actually contained in the list or not. By using such a list, we would be able to obtain a list of prime numbers with existing skeletons for fixed-length lists. However, we cannot make a list of pairs of adjacent prime numbers, because adjacent Boolean-integer pairs are not always adjacent prime numbers. Furthermore, we are forced to write complicated code due to the manipulation of Boolean-integer pairs.

The restriction that the length of a list is fixed mainly comes from an efficiency reason. In existing skeleton libraries for distributed environments, elements in a list are statically distributed at its initialization time. This enables these libraries to be efficiently implemented without extra communications between nodes. The most straightforward way for removing this restriction is to provide new list skeletons, each of which produces within a memory area an output list whose length is different from that of an input list. However, these skeletons are inefficient in many cases, because they need extra memory allocations. Thus, we want new list skeletons with the following features.

- To avoid inefficient memory usage, each skeleton destructively reuse the memory occupied by its input list for its output list that may have different length.
- To avoid inefficient data communications that dissolve uneven data distributions of elements of a list, dynamic and flexible data re-distribution mechanism is necessary in the implementations of the new skeletons.

In this paper, we propose data parallel skeletons for variable-length lists and implement them within a parallel skeleton library called SkeTo [14][16]. The skeletons for variable-length lists enable us to solve a wide range of problems with SkeTo.

The research discussed in this paper makes three main contributions.

1. We analyzed typical patterns of problems that needed variable-length lists and determined which skeletons and operations would be supported in the SkeTo library. Although the new skeletons and operations that were supported were quite common, especially within the community of functional programming languages, they should suffice to describe programs for typical problems.
2. We demonstrated that a block-cyclic representation of a list with a size table effectively enabled a variable-length list to be implemented. This representation was flexible enough to cope with changes in the list's length, and enabled the library to delay the relocation of elements in a list whose elements were unevenly distributed between nodes.
3. Through tests in various experiments, we confirmed the efficiency of our implementation of variable-length lists.

2 Existing List Skeletons in SkeTo

SkeTo (Skeletons in Tokyo) [14][16] is a parallel skeleton library intended for distributed environments such as PC clusters, in which all machines (called *nodes* after this) in the cluster are composed of a single-core CPU. It has three distinguishing features.

1. It provides a set of data parallel skeletons that are based on the theory of Constructive Algorithmics [4] for recursively-defined data structures and functions.
2. It enables users to write parallel programs as if they were sequential, since the distribution, gathering, and parallel computation of data are concealed within constructors of data types or definitions of parallel skeletons,
3. It provides various kinds of data types such as lists (distributed one-dimensional arrays), matrices (distributed two-dimensional arrays), and trees (distributed binary trees).

SkeTo is implemented in C++ with the MPICH library. A list provided by the current version of SkeTo is restricted to be of fixed length. The most important skeletons for lists are **map**, **reduce**, **scan**, and **zip**.

The **map** is a skeleton that applies a function to all elements in a list. The **reduce** is a skeleton that collapses a list into a single value by repeated applications of a certain associative binary operator. The **scan** is a skeleton that accumulates all intermediate results for **reduce**. The **zip** creates a list of pairs of corresponding elements in two given lists of the same length. By letting \oplus be an associative operator, these four skeletons can be informally defined as

$$\begin{aligned}
 \mathbf{map} (f, [x_1, x_2, \dots, x_n]) &= [f(x_1), f(x_2), \dots, f(x_n)], \\
 \mathbf{reduce} ((\oplus), [x_1, x_2, \dots, x_n]) &= x_1 \oplus x_2 \oplus \dots \oplus x_n, \\
 \mathbf{scan} ((\oplus), [x_1, x_2, \dots, x_n]) &= [x_1, x_1 \oplus x_2, \dots, x_1 \oplus x_2 \oplus \dots \oplus x_n], \text{ and} \\
 \mathbf{zip} ([x_1, x_2, \dots, x_n], [y_1, y_2, \dots, y_n]) &= [(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)].
 \end{aligned}$$

Table 1. C++ interfaces of existing list.

List skeletons	
<code>dist_list<A>* map(F& f, dist_list<A>* as)</code>	Map a list
<code>void map_ow(F& f, dist_list<A>* as)</code>	Map a list with overwrite
<code>A reduce(OPLUS& oplus, dist_list<A>* as)</code>	Reduce a list
<code>dist_list<A>* scan(OPLUS& oplus, dist_list<A>* as)</code>	Scan a list
<code>void scan_ow(OPLUS& oplus, dist_list<A>* as)</code>	Scan a list with overwrite
<code>dist_list<std::pair<A,B> >* zip(dist_list<A>* as, dist_list* bs)</code>	Zip a list

List constructors	
<code>dist_list<A>::dist_list(F& f, int size)</code>	Create a list using f
<code>dist_list<A>::dist_list(A* array, int size)</code>	Create a list using array

A list is provided as C++ template class `dist_list<A>`, where `A` is a template parameter. This is implemented as a distributed one-dimensional array. A list is initialized by using its constructor in two ways; the first is given the initial data to be distributed to each corresponding node, and the second is given a generator function to compute the initial values of the elements in the list at each node. Table 1 shows the C++ interfaces of the skeletons and constructors of the lists.

In the current implementation of SkeTo, the elements in lists are equally distributed to each node using block placement. The data placement does not change during computation because no skeletons that can dynamically change a list's length are provided.

3 Design of Variable-Length Lists

3.1 Problems That Need Variable-Length Lists

We classified problems that need variable-length lists into three groups.

1. Problems that leave such elements in a given list that satisfy various conditions.
2. Searching problems in which the number of candidates for solutions may dynamically change.
3. Iterative calculations in which computational loads for all elements in a list lack uniformity.

Examples in the first group include the problem of twin primes discussed in Sec. 1 and the problem where the convex hull of a set of points is computed by using the gift-wrapping algorithm. To solve the twin-primes problem, as we have already stated in Sec. 1 we need to add an element to a list, remove an element from it, and concatenate two lists. To solve the problem of the convex hull, we need to extract points that create the convex hull from a list of points and add them into another list. Since fixed-length lists do not have these operations, we cannot solve these problems.

An example in the second group is the Knight's Tour problem. Given a chess board with $N \times M$ squares, this problem involves finding the numbers of paths for a knight

that visits each square only once. Even though we can represent possible board states during computation with a fixed-length list, we cannot represent new board states after the next move by the knight. This is because each state may generate zero or many next states and therefore the length of the list has to be dynamically changed. To describe a program for this problem, we have to create a list of possible board states after the first k moves by the knight locally on the master node and then distribute these among nodes for further sequential computation on each node. It is a difficult task to create such a program by only using lists of fixed length.

Examples in the third group include calculations of Mandelbrot and Julia sets, in which the computational load at each point on a plane is dramatically different from the other points. When we represent a set of points on a plane with a fixed-length list, we cannot solve this problem efficiently due to load imbalance. If we can remove elements that have already finished their calculations in a list in iterative calculations (e.g., at regular intervals), we can solve these problems efficiently. Unfortunately, we cannot remove elements from a fixed-length list.

3.2 Skeletons and Operations for Variable-Length Lists

To solve the problems in the previous section, we need operations that add elements to a list, remove elements from it, and concatenate two lists to constitute a long list. To achieve these goals, we propose two skeletons, namely, **concatmap** and **filter**, and five operations, namely, **append**, **pushfront**, **pushback**, **popfront**, and **popback**. In addition, we revised the definition of **zip** to accept two lists of different lengths.

The **concatmap** is a skeleton that applies a function to every element in a list and concatenates the resulting lists to generate a flattened list. The **filter** is a skeleton that takes a Boolean function, p , and a list, and leaves elements in the list that satisfy p . The **append** is an operator that concatenates two lists. Note that they do not necessarily have the same length. The **popfront** and **popback** are respective operations that remove an element from the front and the back in a list. The **pushfront** and **pushback** are respective operations that add an element to the front or the back in a list. They are informally defined as

$$\begin{aligned}
 \text{concatmap } (f, [x_1, x_2, \dots, x_n]) &= [x_{11}, x_{12}, \dots, x_{1m_1}, x_{21}, x_{22}, \dots, x_{2m_2}, \\
 &\quad \dots, x_{n1}, x_{n2}, \dots, x_{nm_n}], \\
 &\quad \text{where } f(x_i) = [x_{i1}, x_{i2}, \dots, x_{im_i}], \\
 \text{filter } (p, [x_1, x_2, \dots, x_n]) &= \text{concatmap } (f, [x_1, x_2, \dots, x_n]), \\
 &\quad \text{where } f(x) = \text{if } p(x) \text{ then } [x] \text{ else } [], \\
 \text{append } ([x_1, \dots, x_n], [y_1, \dots, y_m]) &= [x_1, \dots, x_n, y_1, \dots, y_m], \\
 \text{popfront } ([x_1, x_2, \dots, x_n]) &= (x_1, [x_2, x_3, \dots, x_n]), \\
 \text{popback } ([x_1, x_2, \dots, x_n]) &= (x_n, [x_1, x_2, \dots, x_{n-1}]), \\
 \text{pushfront } (v, [x_1, x_2, \dots, x_n]) &= [v, x_1, x_2, \dots, x_n], \text{ and} \\
 \text{pushback } (v, [x_1, x_2, \dots, x_n]) &= [x_1, x_2, \dots, x_n, v].
 \end{aligned}$$

By using these skeletons and operations, we can describe programs for the problems discussed in Sec. 3.1 with variable-length lists. As examples, we have given pseudo-codes for the Knight's Tour problem (Fig. 2) and the Mandelbrot set calculation (Fig. 3).

```

boards = []; //list of solution boards
// add initial board to boards
pushback(initBoard, boards); // increase boards dynamically up to MAXSIZE
while( length(boards) < MAXSIZE ){
    concatmap(NextBoard(), boards); // generate next moves from each boards
}
// search all solutions with depth first order
concatmap(Solve(), boards); // boards is a list of solution boards

```

Fig. 2. Pseudo-code for Knight's Tour problem

```

points = [...]; // list of points
result_points = []; // list of calculation results
for ( int i=0; i<maxForCount; i++ ){
    map(Calc(), points); // progress calculations in small amounts
    // remove elements that have already finished calculation using filter
    end_points = filter(IsEnd(), points);
    append(result_points, end_points); // add them to result_points
}
append(result_points, points); // result_points is a list of calculation results

```

Fig. 3. Pseudo-code for Mandelbrot-set calculation

4 Implementation

4.1 C++ Interfaces and Program Example

We implemented the variable-length lists as a new library of SkeTo. Variable-length lists are provided as a C++ template class called `dist_list<A>`, which has all interfaces of existing fixed-length lists for compatibility. Thus, a fixed-length list is a special case of a variable-length list where the length of the list never changes. Users do not need to change their existing programs based on fixed-length lists to use the new library. Table 2 shows C++ interfaces of skeletons and operations that were introduced in this new library. Note that `push_front`, `push_back`, `pop_front`, and `pop_back` destructively update the input list as a side effect.

Figure 4 shows a concrete C++ program for the twin primes problem. First, we remove every element whose value is not a prime number with `filter_ow` to create a list of prime numbers. Second, to create a list of pairs of adjacent prime numbers, we apply `zip` to two lists; the first is a list of prime numbers and the second is also a list of prime numbers in which the first prime, 2, is removed by `pop_front`. Finally, we remove every pair whose difference is not two by using `filter_ow` to obtain a list of twin primes.

4.2 Data Structures

There are two requirements for the data structures of variable-length lists. First, each node has to know the latest information on the numbers of elements in other nodes to enable programs to access arbitrary elements in the list and to detect load imbalance. Second, we have to reduce the number of times of data relocation as few as possible because this involves enormous overheads in transferring large amounts of data from one node to another.

Table 2. C++ interfaces of the skeletons and operations

List skeletons	
<code>dist_list<A>*</code>	Filter a list
<code>filter(F& f, dist_list<A>* as)</code>	
<code>void filter_ow(F& f, dist_list<A>* as)</code>	Filter a list with overwrite
<code>dist_list<A>*</code>	Concatmap a list
<code>concatmap(F& f, dist_list<A>* as)</code>	
<code>void concatmap_ow(F& f, dist_list<A>* as)</code>	Concatmap a list with overwrite
List operations	
<code>void</code>	Concatenate lists
<code>dist_list<A>::append(dist_list<A>* as)</code>	
<code>void dist_list<A>::push_front(A v)</code>	Add v at front of a list
<code>void dist_list<A>::push_back(A v)</code>	Add v to the back of a list
<code>A dist_list<A>::pop_front()</code>	Obtain an element from the front of a list
<code>A dist_list<A>::pop_back()</code>	Obtain an element from the back of a list

To achieve this end, we introduced *size tables* for the first requirement, and *block-cyclic* placement for the second. A size table has information on the number of elements at each node. When the number of elements changes due to `concatmap` or `filter`, the size table in each node is updated using inter-node communication. Moreover, when we concatenate two lists with `append`, we adopt block-cyclic placement without having to relocate elements on the lists.

Figure 5 shows an example of the initial data structure of a variable-length list, `as`, in which elements in the list have been equally distributed among nodes using block placement. When we remove every element whose value is a multiple of three with `filter_ow`, the number of elements in `as` changes in each node. Thus, the size table at each node is accordingly updated. If the numbers of elements in nodes become too unbalanced, the data in the list are automatically relocated. Returning to the example of

```

dist_list<int>* numbers = new dist_list<int>(SIZE);
dist_list<int>* primes = new dist_list<int>(0);
dist_list<pair<int, int>>* twin_primes;
...add integers, which is more than 1, to numbers...
// make a list of primes using the Sieve of Eratosthenes
int prime;
do {
    prime = numbers->pop_front();
    skeletons::filter_ow(IsNotMultipleOf(prime), numbers);
    primes->push_back(prime);
} while ( val <= sqrt_size );
primes->append(numbers);
// make a list of twin primes from a list of primes
dist_list<int>* dup_primes = primes->clone<int>();
dup_primes->pop_front();
twin_primes = skeletons::zip(primes, dup_primes);
filter_ow(twin_prime, IsCouple());
// twin_primes is solution list

```

Fig. 4. C++ program for twin primes problem

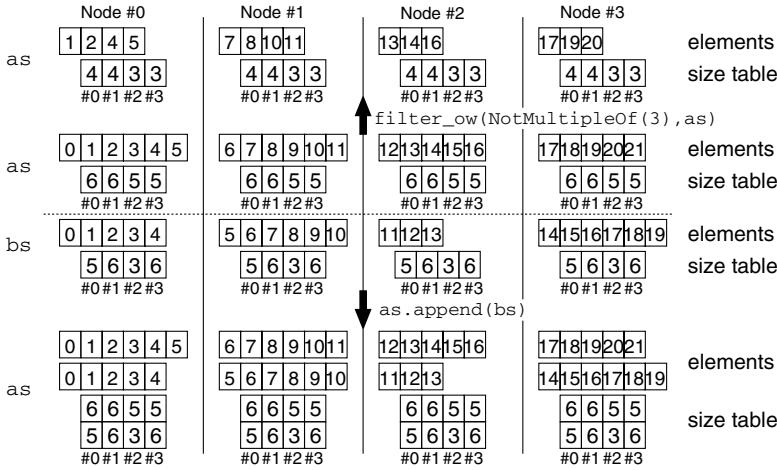


Fig. 5. Data structure of proposed variable-length list

list `as`, when we concatenate `as` and `bs` to make a long list with `append`, we adopt block-cyclic placement without relocating the entire amount of data in the resulting list. After this, we will call part of a list a *block*, whose elements are sequentially allocated on each node, and will call the number of blocks a *block count*. For example, in Fig. 5, we concatenated two lists (`as` and `bs`) whose block counts are both one, and the block count of the resulting list (destructively updated `as`) is two.

4.3 Implementation of Skeletons

We implemented all skeletons based on the data structures described in the previous subsection. The `map`, `concatmap`, and `filter` skeletons simply apply a given function to each block in the local lists at each node. Note that they do not need any inter-node communication. These skeletons do not degrade their performances due to the number of block counts on a list.

The `reduce` and `scan` skeletons need to transfer the results of local calculation from one node to another. The `reduce` calculates local reductions at each node and then folds all local results by using inter-node communication based on the binary-tree structure of the nodes to obtain the final result. The `scan` first applies a local scan to each block at each node and then shares the results by using all-to-all communication. Finally, each node calculates the residual local scan. The overheads for these two skeletons due to inter-node communication increase in proportion to the block counts.

The `zip` skeleton first relocates one of the given lists to make the shapes of the lists the same. It then creates a list of pairs of corresponding elements.

4.4 Data Relocation

In two cases, we have to relocate the data in a list to maintain efficient calculations.

1. When the numbers of elements on each node become too unbalanced.
2. When the block counts of a list reach a large value.

Case 1 is caused by the applications of `concatmap` or `filter`. Case 2 is caused by the concatenation of lists with `append`. This reduces the efficiencies of not only `reduce` and `scan`, but also that of updating the size table.

To detect case 1, we introduce the measure *imbalance* of a list. When the value of *imbalance* becomes greater than a threshold value, t_u , we relocate the data in the list. The *imbalance* of list *as* is defined as:

$$\text{imbalance}(\text{as}) = n \times \max(P_1, \dots, P_n) / (P_1 + \dots + P_n)$$

where $P_i = \sum_{j=1}^m p_{ij}$, n = the number of nodes, m = the block count of *as*, p_{ij} = the number of elements of the j -th block at the i -th node.

Using a rough estimate by letting the value of *imbalance* be r , it takes r times longer to apply a skeleton than where the numbers of elements in each node are completely even. The value of t_u is empirically determined to be 1.5.

For case 2, we relocate the data in a list when its block count becomes greater than a threshold value, t_b . The value of t_b is empirically determined to be 2048.

The relocation for both cases is performed, if necessary, before each skeleton is applied. After relocation, the elements in the list are evenly distributed with block placement, i.e., the block count becomes 1.

5 Experiment

This section describes the effectiveness of the variable-length lists we propose by presenting the results for micro- and macro-benchmarks. The experimental environment we used was a PC cluster system with 16 processors connected through a 1000 BaseT Ethernet. Each PC had an Intel Pentium 4 (3.0 GHz) CPU and 1 GB of main memory. The operating system was Linux kernel 2.6.8., the compiler was GCC 3.3.5 with optimizing level O2, and the MPI implementation was MPICH 1.2.6.

5.1 Micro-benchmark

The efficiency of skeletons and data relocation degrades as the block count of a list increases. To evaluate these overheads, we measured the execution times for applying `map`, `reduce`, and `scan` to an input list of 80,000,000 elements, increasing the block count of the list from 1 to 4000. We used two functions: the first had a short execution time (only a single multiplication) and the second had a long execution time (1000 multiplications). We also measured the running times for the data relocation in a list.

The results are listed in Table 3. Execution times for block count 1 can be regarded as those of the existing SkeTo implementation that supports only fixed-length lists. Thus, each difference between execution times for block count m ($m > 1$) and 1 is the overhead due to the block-cyclic implementation of variable-length lists. Little overhead was observed in the applications of `map` to a list even if its block count was large. When the applied function needed a short execution time, we can see rather large overheads in `reduce` and `scan` to lists with large block counts. These overheads are due to the cost of inter-node communications for transferring the results of local calculations, and, particularly for `scan`, the cost of the residual local scan. In contrast, the overhead of these skeletons is relatively small when the applied function needs a long time for execution. The overhead for data relocation is large. Thus, it would be effective to delay the relocation of data.

Table 3. Results for micro-benchmark

Block count	Execution time (s)						
	1	10	100	1000	2000	3000	4000
Map (short duration)	0.0128	0.0129	0.0128	0.0128	0.0129	0.0130	0.0132
Reduce (short duration)	0.0183	0.0182	0.0183	0.0191	0.0194	0.0197	0.0200
Scan (short duration)	0.0407	0.0408	0.0411	0.0443	0.0484	0.0530	0.0580
Map (long duration)	16.8	16.8	16.8	16.8	16.8	16.8	16.8
Reduce (long duration)	16.9	16.9	16.9	16.9	16.9	16.9	17.0
Scan (long duration)	33.8	33.8	33.8	33.9	33.9	34.0	34.1
Data relocation	–	3.74	4.64	4.67	4.66	4.62	4.72

Table 4. Results for macro-benchmark

Number of nodes	Execution time (s) / Ratio				
	1	2	4	8	16
Twin primes	16.86 / 1.00	8.84 / 0.52	4.52 / 0.27	2.38 / 0.14	1.52 / 0.09
Gift-wrapping method	8.59 / 1.00	4.38 / 0.51	2.21 / 0.26	1.13 / 0.13	0.59 / 0.07
Knight’s tour	11.92 / 1.00	6.05 / 0.51	3.34 / 0.28	2.53 / 0.21	1.22 / 0.10
Mandelbrot set (var. list)	63.4 / 1.00	31.8 / 0.50	16.2 / 0.26	8.2 / 0.13	4.2 / 0.07
Mandelbrot set (fix. list)	61.2 / 1.00	30.6 / 0.50	29.9 / 0.49	20.8 / 0.34	11.9 / 0.19
Julia set (var. list)	60.0 / 1.00	30.1 / 0.50	15.1 / 0.25	7.6 / 0.13	4.1 / 0.07
Julia set (fix. list)	56.7 / 1.00	28.3 / 0.50	21.6 / 0.38	12.3 / 0.22	6.7 / 0.12

5.2 Macro-benchmark

We measured the execution times of the programs to solve the problems discussed in Sec. 3. We provided the following input to each problem: a list of 10,000,000 integers for the twin primes problem, 1,000,000 randomly generated and uniformly distributed points for the convex hull problem, a 5×6 board for the Knight’s Tour, and $1,000 \times 1,000$ coordinates for both the Mandelbrot and Julia set problems with 100 iterative calculations \times 100 times. We also measured the execution times of the programs for the Mandelbrot and Julia set problems using fixed-length lists with 10,000 iterations. The results are shown in Table 4. These results indicate excellent performance in all problems with variable-length lists. The programs for the Mandelbrot and Julia set problems with variable-length lists demonstrated particularly good speedups compared to those with fixed-length lists because there was adequate load balancing.

6 Related Work

Skeletal parallel programming was first proposed by Cole [8] and a number of systems (libraries) have been proposed so far. P3L [3] supports both data parallel and task parallel skeletons. A P3L program has a two-layers structure. Higher skeleton level is written in a functional notation, while lower base language level is described in the C language. Muesli [13] is a C++ library that also supports data parallel and task parallel

skeletons without syntactic enhancements. Both P3L and Muesli offer lists (distributed one-dimensional arrays) and matrices (distributed two-dimensional arrays). Quaff [11] is another skeleton library in C++. It relies on template-based meta-programming techniques to attain high efficiency. However, these three libraries do not support variable-length lists. The latest version of eSkel [59] supports task parallel skeletons for pipelining or master-worker computations, putting emphasis on addressing the issues of nesting of skeletons and interaction between parallel activities. However, it does not support data parallel skeletons for lists like `map` and `reduce`. Lithium [1] is a library written in Java that supports common data and task parallel skeletons. Muskel [2], which is a successor of Lithium, is a full Java library targeting workstation clusters, networks, and grids. They are implemented based on (macro) data flow technology rather than template technology exploited by SkeTo. Calcium [6] is also a Java skeleton library on a grid environment. It mainly focuses on performance tuning of skeletal programs.

Another group of libraries that support distributed arrays contains MCSTL [17], and STAPL [18], each of which is an extension of C++ standard template library (STL) for parallel environments. MCSTL has a distributed fixed array whose target is shared-memory multiprocessor environments. STAPL has `pVector` and `pList`, which correspond to variable-length arrays and lists, and it targets both shared- and distributed-memory environments. However, STAPL does not have operations such as `zip` and `concatmap`, since they only provide the same operations as STL. Data Parallel Haskell [7,12] offers distributed nested lists in which we can apply `filter` and `concatmap` to lists and concatenate lists as well as our variable-length lists. However, it only targets shared-memory multiprocessor environments.

7 Conclusion

We proposed parallel skeletons for variable-length lists and their implementation within a parallel skeleton library called SkeTo. A variable-length list enables us to dynamically increase/decrease the number of elements and thus solve a wide range of problems including those of twin primes, Knight's tour, and Mandelbrot set calculation. Our implementation adopted a block-cyclic representation of lists with size tables, whose efficiency was proved through tests conducted in various experiments. We intend to include the variable-length lists and related skeletons presented in this paper in future releases of SkeTo.

Acknowledgments. We wish to thank Masato Takeichi, Kenetsu Hanabusa, Zhenjiang Hu, Kiminori Matsuzaki, and other POP members in Tokyo for their fruitful discussions on the SkeTo library. This work was partially supported by Grant-in-Aid for Scientific Research (20500029) from the Japan Society of the Promotion of Science.

References

1. Aldinucci, M., Danelutto, M., Teti, P.: An Advanced Environment Supporting Structured Parallel Programming in Java. *Future Gener. Comput. Syst.* 19(5), 611–626 (2003)
2. Aldinucci, M., Danelutto, M., Dazzi, P.: Muskel: an Expandable Skeleton Environment. *Scalable Computing: Practice and Experience* 8(4), 325–341 (2007)

3. Bacci, B., Danelutto, M., Orlando, S., Pelagatti, S., Vanneschi, M.: P3L: A Structured High Level Programming Language, and its Structured Support. *Concurrency: Pract. Exper.* 7(3), 225–255 (1995)
4. Backhouse, R.: An Exploration of the Bird-Meertens Formalism. In: STOP Summer School on Constructive Algorithmics, Abelard (1989)
5. Benoit, A., Cole, M., Gilmore, S., Hillston, J.: Flexible skeletal programming with eSkel. In: Cunha, J.C., Medeiros, P.D. (eds.) *Euro-Par 2005. LNCS*, vol. 3648, pp. 761–770. Springer, Heidelberg (2005)
6. Caromel, D., Leyton, M.: Fine tuning algorithmic skeletons. In: Kermarrec, A.-M., Boug  , L., Priol, T. (eds.) *Euro-Par 2007. LNCS*, vol. 4641, pp. 72–81. Springer, Heidelberg (2007)
7. Chakravarty, M.M.T., Leshchinskiy, R., Jones, S.L.P., Keller, G., Marlow, S.: Data Parallel Haskell: A Status Report. In: 2007 ACM Workshop on Declarative Aspects of Multicore Programming (DAMP 2007), pp. pp. 10–18. ACM Press, New York (2007)
8. Cole, M.: Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation. *Research Monographs in Parallel and Distributed Computing*, Pitman (1989)
9. Cole, M.: Bringing Skeletons out of the Closet: a Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Comput.* 30(3), 389–406 (2004)
10. Darlington, J., Field, A.J., Harrison, P.G., Kelly, P.H.J., Sharp, D.W.N., Wu, Q., While, R.L.: Parallel Programming Using Skeleton Functions. In: Reeve, M., Bode, A., Wolf, G. (eds.) *PARLE 1993. LNCS*, vol. 694, pp. 146–160. Springer, Heidelberg (1993)
11. Falcou, J., S  rot, J., Chateau, T., Laprest  , J.T.: QUAFF: Efficient C++ Design for Parallel Skeletons. *Parallel Comput.* 32(7), 604–615 (2006)
12. Jones, S.L.P., Leshchinskiy, R., Keller, G., Chakravarty, M.M.T.: Harnessing the Multicores: Nested Data Parallelism in Haskell. In: IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2008 (2008)
13. Kuchen, H.: A skeleton library. In: Monien, B., Feldmann, R.L. (eds.) *Euro-Par 2002. LNCS*, vol. 2400, pp. 620–629. Springer, Heidelberg (2002)
14. Matsuzaki, K., Emoto, K., Iwasaki, H., Hu, Z.: A Library of Constructive Skeletons for Sequential Style of Parallel Programming. In: 1st International Conference on Scalable Information Systems, InfoScale 2006 (2006)
15. Rabhi, F.A., Gorlatch, S. (eds.): *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, Heidelberg (2002)
16. SkeTo Project, <http://www.ipl.t.u-tokyo.ac.jp/sketo/>
17. Singler, J., Sanders, P., Putze, F.: MCSTL: The multi-core standard template library. In: Kermarrec, A.-M., Boug  , L., Priol, T. (eds.) *Euro-Par 2007. LNCS*, vol. 4641, pp. 682–694. Springer, Heidelberg (2007)
18. Tanase, G., Bianco, M., Amato, N.M., Rauchwerger, L.: The STAPL pArray. In: 2007 Workshop on Memory performance: Dealing with Applications, systems and architecture (MEDEA 2007), pp. 73–80. ACM Press, New York (2007)

STKM on SCA: A Unified Framework with Components, Workflows and Algorithmic Skeletons

Marco Aldinucci¹, Hinde Lilia Bouziane², Marco Danelutto³,
and Christian Pérez⁴

¹ Dept. of Computer Science, University of Torino
aldinuc@di.unito.it

² LIP/Lyon 1 University, ENS Lyon
hinde.bouziane@ens-lyon.fr

³ Dept. of Computer Science, University of Pisa
marcod@di.unipi.it

⁴ LIP/INRIA, ENS Lyon
christian.perez@inria.fr

Abstract. This paper investigates an implementation of STKM, a Spatio-Temporal sKeleton Model. STKM expands the Grid Component Model (GCM) with an innovative programmable approach that allows programmers to compose an application by combining component, workflow and skeleton concepts. The paper deals with a projection of the STKM model on top of SCA and it evaluates its implementation using Tuscany Java SCA. Experimental results show the need and the benefits of the high level of abstraction offered by STKM.

1 Introduction

Quite a large number of programming models have been and are currently proposed to support the design and development of large-scale distributed scientific applications. These models attempt to offer suitable means to deal with the increasing complexity of such applications as well as with the complexity of the execution resources (e.g. those in grids and/or clouds) while attempting to ensure efficient execution and resource usage. However, existing models offer peculiar features that actually make them suitable for specific kind of applications only. A current challenge is still to be able to offer a suitable programming model that easily and efficiently supports multi-paradigm applications.

Three different programming models have recently been considered to support large-scale distributed scientific applications: software components, workflows and algorithmic skeletons. All these models follow an assembly/composition programming principle, which is becoming a widely accepted methodology to cope with the complexity of the design of parallel and distributed scientific applications. *Software components* promote code reuse . Components can be composed to build new and more complex components so that applications are component assemblies. Such assemblies are usually defined at compile time, although most component frameworks provide mechanisms that can be used to implement dynamic assemblies at run time. An assembly completely determines spatial interactions among components, and therefore it is referred to as *spatial* composition. Due to these features, component models are suitable

to support strongly coupled compositions. *Workflow* models have been mainly developed to support composition of independent programs (usually loosely coupled and named tasks) by specifying temporal dependencies among them (and defining a *temporal composition*, in fact), to support efficient scheduling onto available resources (e.g. sites, processors, memories) [2]. Last but not least, *algorithmic skeletons* have been introduced to support typical parallel composition patterns (skeletons) [3]. Skeleton composition follows precise rules and skeleton applications are (well formed) skeleton assemblies. The composition style is mostly spatial, as in the software component case, although skeletons processing streams of input data sets usually present several embedded temporal composition aspects. Skeleton assembly “structures” can be exploited to provide automatic optimization/tuning for efficient execution on targeted resources.

In summary, each one of these three models is worth being used in some particular circumstances, but nevertheless all their properties seem to be relevant and worth to be considered in a single model. In [4], we discussed STKM (*Spatio-Temporal sKeleton Model*), a single programming framework providing programmers with components, workflows and skeletons. STKM allows these three abstractions to be mixed in arbitrary ways in order to implement complex applications. While in [4] we explored the *theoretical background* of STKM, in this paper, we concentrate on the problems related to the *implementation* of STKM components, realized as an extension of GCM (*Grid Component Model* [5]) components built on top of SCA (*Service Component Architecture* [6]).

This paper is organized as follows: Section 2 outlines relevant related work, Section 3 sketches the main features of STKM. SCA implementation design of STKM and related issues are presented in Section 4 while experimental results showing the feasibility of the whole approach are discussed in Section 5. Section 6 concludes the paper.

2 Background and Related Work

Skeleton based programming models allow application programmers to express parallelism by simply instantiating – and possibly nesting – items from a set of predefined patterns, the skeletons, that model common parallelism exploitation patterns [3]. Typical skeletons include both stream parallel and data parallel common patterns [7,8,9]. Programming frameworks based on algorithmic skeletons achieve a complete and useful separation of concerns between application programmers (in charge of recognizing parallelism exploitation patterns in the application at hand and of modeling them with suitable skeletons) and system programmers (in charge of solving, once and for all, during skeleton framework design and implementation, the problems related to the efficient implementation of skeletons and of skeleton composition). In turn, this separation of concerns supports rapid application development and tuning, allows programmers without specific knowledge on parallelism exploitation techniques to develop efficient parallel applications, and eventually supports seamless (from the application programmer perspective) porting of applications to new target architectures.

Skeleton technology has recently been adopted in the component based programming scenario by developing (composite) components modeling common parallelism exploitation patterns and accepting other components as parameters to model the skeleton inner computations [10,11]. To support automatic adaptation of skeleton component

execution to highly dynamic features of target architectures such as grids, autonomic management has been eventually combined with skeleton based modeling in the *behavioural skeletons* [12]. The result is that behavioural skeletons – or a proper nesting of behavioural skeletons – can be simply instantiated to obtain fully functional, efficient parallel applications with full, autonomic auto tuning of performance concerns.

Component based skeleton frameworks provide all the optimizations typical of algorithmic skeletons. Behavioural skeletons add the autonomic tuning of non-functional concerns. However, no support has been provided, as far as we know, to support skeleton components in workflows. Skeletons in workflows would allow to express computations as temporal compositions while preserving the possibility to optimize parallel workflow stages according to the well-known results of the skeleton technology.

3 STKM

STKM (*Spatio-Temporal sKeleton Model* [4]) extends STCM [13], a *Spatio-Temporal Component Model* merging component and workflow concepts, with (behavioural) skeleton support. This extension promotes more simplicity of design and separation of functional concerns from non-functional ones (management of components life cycle, parallelism, etc.). It also promotes the portability of applications to different execution contexts (resources). For that, a level of abstraction is offered to allow a designer to express the functional behaviour of an application through its assembly. The behaviour expressiveness is exploited by an STKM framework to adapt the application to its execution context. This section outlines the unit of composition of STKM, its assembly model and a suitable approach to manage the assembly by the framework.

An STKM *component* is a combination of a classical software component and task (from workflows) concepts. As shown in Figure 1, a component can define *spatial* and/or *temporal* ports. Spatial ports are classical component ports. They express interactions between components concurrently active [13]. Temporal ports (input/output) behave like in a workflow, instead. They express data dependences between component tasks and then an execution order of these tasks. Like in a workflow, components in-

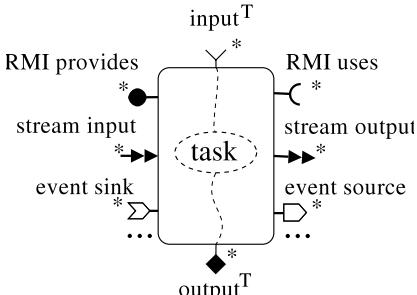


Fig. 1. An STKM component. T: refers to temporal ports. Other ones are spatial.

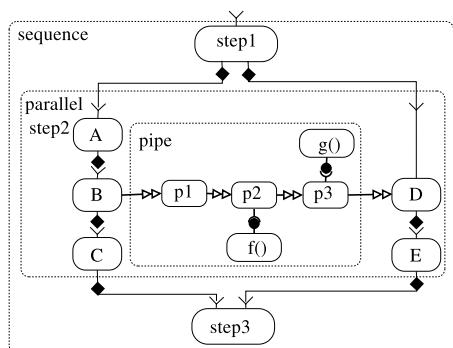


Fig. 2. Example of an STKM assembly

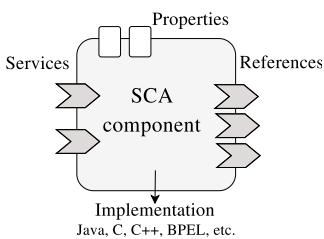
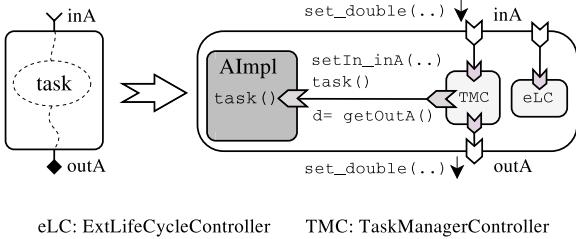
volved in a temporal relation can be instantiated when both control and data flows reach them [13]. The difference is that the life cycle of an STKM component may be longer than the completion of a task in a workflow.

An STKM *assembly* describes spatial and temporal dependencies between components. Spatial dependencies are drawn by spatial port connections. Temporal dependencies are drawn by a data flow (input and output port connections) and a control flow (using constructs like sequences, branches (*if* and *switch*), loops (*for* and *while*), etc.). In addition, STKM defines constructs dedicated to skeleton-based parallel paradigms. These constructs are particular composite components representing skeleton schemes (pipe, farm, functional replication, etc.). They can be composed with components and/or other skeletons at different levels of an assembly. This ability preserves the pragmatic of skeletons. Figure 2 shows a graphical STKM assembly example of a sequence (step1; step2; step3), where step2 is a parallel composition of sequences (A; B; C) and (D; E). A spatial dependency is drawn between these sequences, as component D consumes the results of the (pipe) skeleton construct whose inputs are produced by B. The figure also shows that skeleton elements (the pipe stages) may express dependences with other components (those providing *f* and *g*). These further components belong to the same assembly of the pipeline skeleton. In general, skeleton assemblies follow a fixed schema, the one typical of the parallel pattern implemented by the skeleton. In this case, the assembly is a linear assembly of stages. Each stage, may involve any number of components in a sub-assembly, provided that it has an input (output) stream port to be connected to the previous (next) pipeline stage.

To be executed, an STKM application needs to be transformed into a concrete assembly, that may include a specific engine. This engine represents *de facto* STKM run time. This run time comprehends the mechanisms needed to support both skeleton and workflow aspects of STKM components. Therefore, it differs in the way STKM components are processed to produce the concrete component assembly from other systems that only concentrate on skeletons [14] or workflows [15]. The concrete assembly may introduce non-functional concerns like data flow management or the hidden part of skeletons implementations (cf. Section 4.1). The STKM engine manages the life cycle of components and the execution order of tasks. During the transformation, STKM benefits from the expressive power of an assembly to exploit the maximum parallelism from a part or from the whole assembly, and to adopt an adequate scheduling policy depending on actually available execution resources. Thus, both explicit (using skeleton constructs) and implicit parallelisms can be considered. An example for the last case is the mapping of a *forAll* loop to a functional replication skeleton in which the workers are the body of the loop. Such a mapping should allow STKM to take benefits from already existing (behavioural) skeleton management mechanisms able to deal with performance [16] concerns, security [17], fault tolerance [18], etc.

4 An SCA Based Implementation of STKM

This paper aims at evaluating the feasibility of the STKM model. SCA (*Service Component Architecture* [6]) is used to investigate whether porting STKM concepts to the Web Service world is as effective as porting other GCM concepts, as shown in [19]. As

**Fig. 3.** An SCA component**Fig. 4.** From an STKM component to an SCA one

STCM was devised as an extension of GCM [13], it was natural to explore the feasibility of STKM in the same direction.

SCA is a specification for programming applications according to a *Service Oriented Architecture (SOA)*. It aims to enable composition of services independently from the technologies used to implement them and from any SCA compliant platform. SCA specifies several aspects: assembly, client and component implementation, packaging and deployment. The *assembly* model defines a component as a set of ports named *services* and *references* (Figure 3). Ports can be of several kinds such as Java, OMG IDL (*Interface Definition Language*), WSDL (*Web Services Description Language*), etc. They allow message passing, Web Services or RPC/RMI based communications. The interoperability between communicating components is ensured through dedicated *binding* mechanisms. The assembly of components may be hierarchical. The hierarchy is abstract, thus allowing to preserve encapsulation and simplifying assembly process. The *client and implementation* model describes a service implementation for a set of programming languages. Several means are used: annotations for Java/C++ or XML extensions for BPEL [20], etc. These means permit the definition of services, properties, and meta-data like local or remote access constraints associated to a service. Last, the *packaging and deployment* model describes the unit of deployment associated to a component. For deployment concerns, an SCA platform is free to define its own model.

4.1 Mapping STKM Concepts on SCA

Two main issues arise when implementing STKM on SCA: the projection of the user view of an STKM component to an SCA based implementation, and the management of an STKM assembly at execution. This section discusses these two issues.

STKM Component and Ports. As mentioned earlier, an SCA based implementation of GCM components has been already proposed [19]. It is based on mapping components, client/server ports, controllers and implementations on a set of services/references, SCA components and SCA implementations. This paper does not detail this projection. It focuses on describing a projection of the concepts added by STKM, i.e. temporal ports (inherited from STCM) and skeleton constructs.

An STKM component is mapped to an SCA composite component and temporal ports to a set of services/references and controllers as illustrated in Figure 4. A component appears as a service provided by the user implementation. Temporal ports are

mapped to a set of services provided/used by a transparent control part of the component. This control part (components TMC and eLC in Figure 4) is responsible to manage input and output data availability and task executions. Details about this management can be found in [13]. Note that realizing controllers using components is not a new idea. This promotes composability and code reuse.

STKM skeleton constructs, which are also components, are projected with a similar approach. The difference is that an implementation of a skeleton may consider additional non-functional elements like managers for behavioural skeletons. A simplified example is shown in Figure 5. The figure represents a functional replication behavioural skeleton. Components MGR (manager), E (emitter) and C (collectors) belong to the non-functional part of the skeleton.

Thus, a simple projection of STKM components to SCA components is possible thanks to the full support of RPC/RMI ports and hierarchical composition in SCA.

STKM Assembly. Not only STKM makes an assembly more explicit with respect to the behaviour of an application, but it also aims to enable automatic and dynamic modifications of the assembly. Thus, it is not sufficient to have just a projection of components as described above. It is also necessary to introduce mechanisms to deal with data transfer between dependent components, to order task execution according to both data flow and control flow, and to manage the life cycle of components. Several solutions can be proposed. This paper presents a simple solution, based on a distributed data transfer and a centralized orchestration engine.

With respect to *data management*, the data transfer between data flow connected components can be direct if both components simultaneously exist or it can be indirect through a proxy component, in case the target component does not exist yet. Figure 7 illustrates how such a proxy can be introduced between two components of the sequence

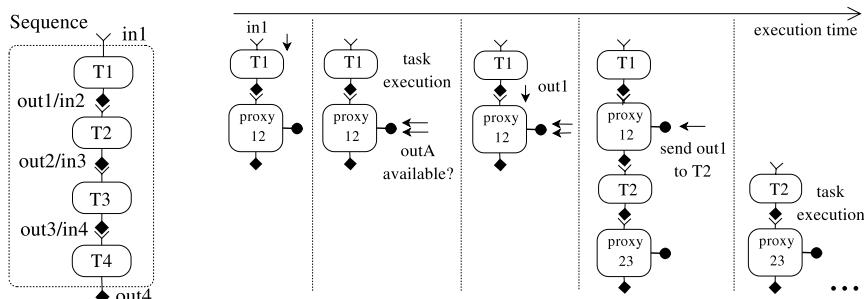


Fig. 6. An STKM example (user view)

Fig. 7. Data transfer through proxy components and dynamic changes of the sequence assembly. The STKM, rather than SCA, notation for ports is still used for simplicity.

assembly of Figure 6. During the execution, `proxy12` receives the output of `T1` and waits a request to send it to `T2` (from the STKM engine, see paragraph below). The availability of the proxy is assumed to be at the responsibility of the transformation engine, which is out of the scope of this paper.

The STKM *engine* is an SCA client program. It contains the sequence of actions which create/destroy components, connect/disconnect ports, manage data availability/transfer and the order of tasks executions. These actions are deduced from the behaviour expressed by the STKM assembly. Figure 7 shows a part of the dynamic assembly evolution of Figure 6 managed by a naive STKM engine which creates components when the data/control flow reaches them. Details about this engine can be found in [21].

4.2 Usage of Tuscany Java SCA

We base the proposed SCA implementation on Tuscany Java SCA Version 1.2.1 [22]. Tuscany is still under development but it provides a preliminary support for distributed executions. However, some features affecting the support of some STKM requirements are not yet supported. The more relevant missing features are related to the dynamicity of an assembly. In Tuscany 1.2.1, to dynamically add/remove components, there are mainly two approaches. The first one is based on dynamic reconfiguration of contributions (components packages). However, it requires to stop the application execution¹. The second approach is based on the addition/removal of *nodes*. A *node* is a process hosting one or more component instances on a given execution resource. The approach does not require to suspend an application execution. However, Tuscany requires nodes to be statically defined. We select this approach and make all decisions static.

SCA specification does not offer an API to connect/disconnect references to services. However, it provides an API to allow passing service references. Using this API, we specified connection/disconnection operations. These operations are associated to a port and exposed as a service implemented by the non-functional part of a component. Note that the service passed by reference is accessed using a Web Service binding protocol in the used Tuscany version.

The advantage of our approach to overcome Tuscany lacking or non-properly supported features, is that no modification in the distribution was needed. However, as discussed in next section, some experiment results are affected by the adopted solutions. For the purpose of this paper, however, this does not prevent the possibility to demonstrate the benefits of STKM and its feasibility on a Web Service based environment.

5 Evaluation

To evaluate the benefits of STKM, this section discusses the performance of the proposed SCA based implementation. We developed an application according to different compositions: sequence, loop, pipeline and nested composition of pipeline and farm/functional replication skeleton constructs, such that we could experiment various

¹ Efforts have been made to overcome this limitation [23]. Its feasibility was proved using Tuscany 1.1.

	Time in s
Remote node Launching	45.56
Programmed port connection	3.20

Fig. 8. Average of times to deploy and connect components

RTT in ms	Intra-node Inter-comp.	Inter-Node Intra-host	Inter-Node Inter-host
Default protocol	0.076	20.35	20.17
WS protocol	22.66	24.23	24.11

Fig. 9. Round Trip Time (RTT) in ms on a local Ethernet network for different situations. WS: Web Service

execution scenarios for a same application with respect to different execution contexts. The application is synthetic and its parameters (amount of involved data, relative weight of computation steps, etc.) have been tuned to stress the different features discussed.

The projection from STKM assembly to SCA components and an STKM engine was manually done using Tuscany Java SCA version 1.2.1 and Java 1.5. The deployment of components was done manually by using `scp` and their creation was done at the initiative of the engine by using `ssh`. Experiments have been done on a cluster of 24 Intel Pentium 3 (800 MHz, 1 GB RAM), running Linux 2.4.18 and connected by a 100 MBit/s switched Ethernet.

5.1 Metrics

We first evaluated basic metrics, including overheads of dynamic life cycle management – nodes launching and port connections – and inter component communications. The results are displayed in Figure 8 and Figure 9. First, it must be pointed out that starting a node is an expensive operation. This is due to remote disk accesses, done through NFS, required to launch a JVM and to load a node process by the used common-daemon library². The time is also stressed by the significant usage of *reflection* and dynamic class loading by Tuscany as well message exchanges between a node and *an SCA domain* (application administrator) for services registry/checking. Second, the time necessary to connect ports through the reference passing mechanism (Section 4.2) is explained by the serialization/deserialization activity needed to pass a reference, and by multiple *reflection* calls. Globally, the overheads stem from the framework and from “tricks” explained in Section 4.2. The round trip times are measured for an empty service invocation for two binding protocols (default “unspecified” SCA and WS) and various placements of components. As explained in Section 4.2, the WS protocol is used for services passed by reference. The results show that the choice of a binding protocol affects communication times. There is a quite important software overhead, that makes network overhead almost negligible in next experiments. This is not a surprise as SCA specification addresses this issue and claims that SCA is more adequate for coarse grain codes or wide area networks.

5.2 Benefits of Resources Adaptation Capability: A Sequence Use Case

This section studies the impact on performance when mapping an STKM assembly to different concrete assemblies. A sequence of 4 tasks (Figure 6) is mapped to the configurations presented in Figure 10. This leads to two concrete assemblies for the abstract

² Apache commons: <http://commons.apache.org/daemon>

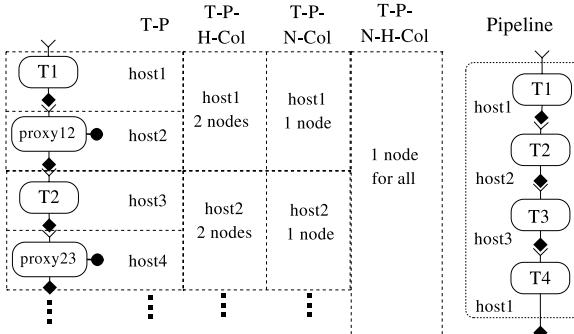


Fig. 10. Configurations used in Figure 11. T: Task in component, P: Proxy, H: Host, N: Tuscany Node, Col: Collocation

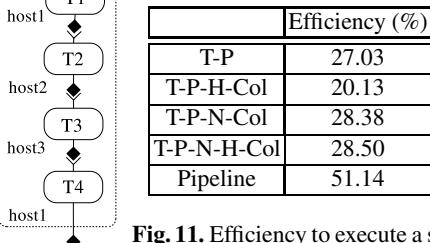


Fig. 11. Efficiency to execute a sequence of 4 tasks. The execution time of each task is 20s. The life cycle of a component is delimited by input/output data availability

one shown in Figure 6 and a given placement of components. For the assembly on the left part of Figure 10, the execution and life cycle management of components follows the principle presented in Section 4.1. On the right part, the sequence is mapped to a pipeline composition. In this case, all concerned components are deployed on different hosts and connected by the engine *before* the execution of the first task.

Figure 11 reports the execution efficiency in terms of the percentage of the whole execution time (including remote (Tuscany) node creation, component instantiation and port connections) spent in the sequence computation. Without surprise, the Pipeline configuration leads to a more efficient execution. This will remain true even if the overheads are lower for coarser grain codes. However, the performance criterion may be not sufficient to choose a particular configuration. Other criteria, such as resource usage, could also be taken into account. Therefore, the Pipeline configuration may cause an over-consumption of resources. That may be problematic when using shared resources, like Grids. Other configurations may offer the ability to optimize resource usage with efficient scheduling policies. The objective of this paper is not to study such policies, so they are not treated and only base cases are tested.

The T-P-N-H-Col configuration behaves better because all components are executed in the same process. Also, the lazy instantiation of components in Tuscany reduces the number of concurrent threads. However, components in a sequence may require different nodes because of execution constraints on memory or processor speed. In such a case, the T-P-N-Col configuration may be more suitable. For the remainder of this section, we selected this configuration.

5.3 Need for Form Recognition

STKM promotes the ability to recognize implicit parallelism forms from an assembly and to exploit them to improve the performance when possible. For that, the approach is to map recognized parallelism forms to a composition of skeleton constructs. This section illustrates the benefits of this approach through a sequential *for* loop. The body of this loop is the sequence shown in Figure 6. The sequence is assumed to be stateless

Loop-Opt

```

STKM engine
  // The engine retrieves the input set of
  // data of the forAll loop to manage it.
  ... // create T1-proxy12
  for (int i = 0; i < data.size(); i++)
    T1.set_double(data[i]);

  // check first output and create T2-proxy21
  // send the output on T2
  // check first output and create T3-proxy34
  // send the output on T3
  ...
  for (int i = 0; i < data.size(); i++)
    ... get_out();

  // remove T1-P1 .... T4-proxy4

```

Pipe skeleton

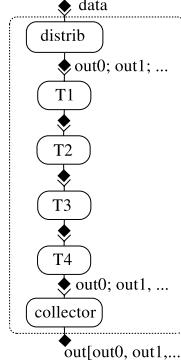


Fig. 12. Overview of the configurations used for executing the *For* loop in a parallel way

	Efficiency (%)
data size: 10x	
Loop	28.40
Loop-Opt	56.16
Pipe	76.96
data size: 100x	
Loop	28.40
Loop-Opt	90.68
Pipe	95.90

Fig. 13. Results of executing a loop according to different configurations. The loop body is a stateless sequence of 4 tasks of 20s

and to be executed for each element of an array of data (input of the loop). The output of the loop is the collection of all the outputs of the different iterations. To execute such a loop, let us study three configurations: *Loop*, *Loop-Opt* and *Pipe*.

In the *Loop* configuration, no parallelism form is recognized. All iterations are executed sequentially and the sequence is mapped to the T-P-Node-Col configuration shown in Figure 10. The *Loop-Opt* configuration exploits the independence of the iterations and the fact that components are stateless to make use of a pipelined execution. The concrete assembly is the same as for the *Loop* configuration. The difference occurs in the management of the data flow and the life cycle of components as shown in Figure 12. As can be noted, the STKM engine is responsible to split/collect the loop input/output data. The pipelined execution is done thanks to the control part of components. This part is responsible to queue received data and to ensure one task execution at a time and the order of treated data (STCM specific behaviour). Note that a component T_i is instantiated once the first output of T_{i-1} is available and removed after the loop execution. The *Pipe* configuration projects the loop to a pipeline skeleton construct. The result is shown on the right part of Figure 12. It introduces two components: *distrib* and *collector* responsible to respectively split (collect) the loop input (output) data into several (one set of) data. All components are instantiated when the control flow reaches the loop and destroyed after retrieving the loop result.

Figure 13 reports results for two different data set sizes. The measures include the overhead related to the life cycle management. Several conclusions can be drawn. First, the *Pipe* configuration presents a more efficient execution. In fact, it is not sufficient to have a pipelined execution (*Loop-Opt*) to reach better performance: an *efficient* implementation of a pipeline, such as one of those available exploiting assessed skeleton technology, is also needed. Second, the overhead of life cycle management can as usual be hidden with longer computation time. Third, in order to achieve efficient execution and management it is necessary to consider the behaviour of the global composition, i.e. combined structures, in an assembly when mapping on a concrete assembly.

	Global time (in s)
Without FR ^a	3105
Farm: 3 workers	1182
FR: dynamic addition of workers	1409

Fig. 14. Pipeline step parallelization using a farm construct. The pipeline is composed of 4 tasks. The execution time of each task is (in order): 10s, 30s, 5s and 5s. Step 3 and 4 are collocated for load balancing. The number of the pipeline input data is 100.

^a FR: Functional Replication

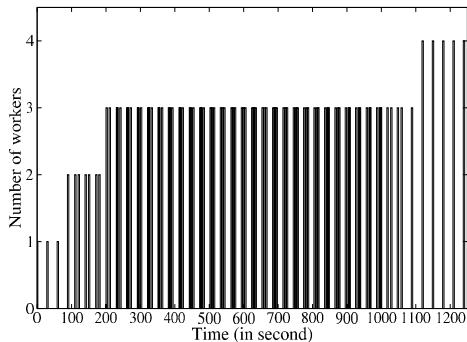


Fig. 15. Dynamic management of workers in a behavioural farm skeleton construct. The farm construct is used to parallelize the second step of the pipeline construct of figure 14.

5.4 Need for Efficient Behavioural Skeleton Management

This section presents experiments illustrating the advantage that behavioural skeletons should offer in the context of STKM. Experiments are relative to the execution of a pipeline composition. The mapping of this composition on a concrete assembly is directed by two criteria: performance and resource usage. For a pipeline, improving these criteria is usually achieved by load-balancing its stages, collocating stages and/or parallelizing bottleneck stages (e.g. by integrating a functional replication skeleton).

The *stage collocation* experiment compares the execution of 4 pipelined tasks with and without applying load balancing. The computation times of the tasks are, in order: 10s, 15s, 5s and 20s. To load balance the stages, the approach was to collocate the second and third stages to be executed on a same host. The efficiency is 96.04% for the collocation scenario and 95.92% without collocation. Thus, it is possible to improve resource usage while preserving performance. While the load balancing was done manually for this experiments, a behavioural skeleton implementation is expected to do it automatically, by monitoring the task execution times and then adequately “grouping” pipeline stages to match the pipeline ideal performance model.

The *stage parallelization* experiment compares the execution of a pipeline composition according to two load-balancing approaches. The pipeline has 4 stages of durations 10s, 30s, 5s and 5s. The objective is to parallelize the execution of a costly pipeline step, here 30s. In addition to the possibility to collocate the third and fourth stages, the second stage is mapped a) to a farm skeleton (Figure 5) with a static number of workers – three in this case – or b) to a “replication” behavioural skeleton (Figure 5) with autonomic and dynamic addition of workers [12]. The skeleton manager implements a simple adaptation policy which adds a worker if the output frequency of the skeleton is less than a given value, 10s, in this case. The results are shown in Figure 14 and Figure 15. As expected, the whole execution time is improved – by a 2.6 factor – when

using a farm with static number of workers. For dynamic workers addition, the number of workers reaches 4 with an improvement of 2.2 instead of 2.6. This is due to the adaptation phase overhead and to the fact that adaptation does not rebalance tasks already in the workers queues. The experiment illustrates the feasibility of realizing a behavioural skeleton in STKM as well as the ability to preserve the advantages of such constructs.

6 Conclusion and Future Works

STKM is a model that aims to increase the abstraction level of applications with enabling efficient executions. Its originality is to unify within a coherent model three distinct programming concepts: components, workflows and skeletons. This paper evaluates its feasibility and its benefits. The proposed mapping introduces a set of non-functional concerns needed to manage an STKM assembly; concerns that can be hidden to the end user and that can be used for execution optimizations. Hand-coded experiments show that STKM can lead to both better performance and resource usage than a model only based on workflows or skeletons.

Future works are fourfold. First, model driven engineering techniques should be considered for implementing STKM to automatize the assembly generation and existing component based behavioural skeleton implementations [19]. Second, investigations have to be made to understand which assembly has to be generated with respect to the resources and to criteria to optimize. Third, techniques to optimize an application are also needed. Fourth, a validation on real word applications [4] has to be done. For that, it should be worth investigating the usage of recent SCA implementation overcoming part of the limitations of the one we used here (e.g. the recently available Tuscany version 1.4).

References

1. Szyperski, C., Gruntz, D., Murer, S.: Component Software - Beyond Object-Oriented Programming, 2nd edn. Addison-Wesley/ACM Press (2002)
2. Fox, G.C., Gannon, D.: Special issue: Workflow in grid systems: Editorials. *Concurr. Comput.: Pract. Exper.* 18(10), 1009–1019 (2006)
3. Cole, M.: Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing* 30(3), 389–406 (2004)
4. Aldinucci, M., Danelutto, M., Bouziane, H.L., Pérez, C.: Towards software component assembly language enhanced with workflows and skeletons. In: Proc. of the ACM SIGPLAN compFrame/HPC-GECO workshop on Component Based High Performance, pp. 1–11. ACM, New York (2008)
5. CoreGRID NoE deliverable series, Institute on Programming Model: Deliverable D.PM.04 – Basic Features of the Grid Component Model, http://www.coregrid.net/mambo/images/stories/Deliverables/d_pm_04.pdf (assessed, February 2007)
6. Beisiegel, M., et al.: SCA Service Component Architecture - Assembly Model Specification, version 1.0. TR, Open Service Oriented Architecture collaboration (OSOA) (March 2007)
7. Kuchen, H.: A skeleton library. In: Monien, B., Feldmann, R.L. (eds.) Euro-Par 2002. LNCS, vol. 2400, pp. 620–629. Springer, Heidelberg (2002)

8. Benoit, A., Cole, M., Gilmore, S., Hillston, J.: Flexible skeletal programming with eSkel. In: Cunha, J.C., Medeiros, P.D. (eds.) Euro-Par 2005. LNCS, vol. 3648, pp. 761–770. Springer, Heidelberg (2005)
9. Caromel, D., Leyton, M.: Fine Tuning Algorithmic Skeletons. In: Kermarrec, A.-M., Bougé, L., Priol, T. (eds.) Euro-Par 2007. LNCS, vol. 4641, pp. 72–81. Springer, Heidelberg (2007)
10. Aldinucci, M., Campa, S., Coppola, M., Danelutto, M., Laforenza, D., Puppin, D., Scarponi, L., Vanneschi, M., Zocco, C.: Components for high performance Grid programming in Grid.it. In: Proc. of the Intl. Workshop on Component Models and Systems for Grid Applications, Saint-Malo, France. CoreGRID series, pp. 19–38. Springer, Heidelberg (2005)
11. Gorlatch, S., Duennebecker, J.: From Grid Middleware to Grid Applications: Bridging the Gap with HOCs. In: Future Generation Grids. Springer, Heidelberg (2005); selected works from Dagstuhl 2005 FGG workshop 2005
12. Aldinucci, M., Campa, S., Danelutto, M., Vanneschi, M., Dazzi, P., Laforenza, D., Tonello, N., Kilpatrick, P.: Behavioural skeletons in GCM: autonomic management of grid components. In: Baz, D.E., Bourgeois, J., Spies, F. (eds.) Proc. of Intl. Euromicro PDP 2008: Parallel Distributed and network-based Processing, Toulouse, France, pp. 54–63. IEEE, Los Alamitos (2008)
13. Bouziane, H.L., Pérez, C., Priol, T.: A software component model with spatial and temporal compositions for grid infrastructures. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 698–708. Springer, Heidelberg (2008)
14. Falcou, J., Sérot, J.: Formal Semantics Applied to the Implementation of a Skeleton-Based Parallel Programming Library. In: Parallel Computing: Architectures, Algorithms and Applications. NIC, vol. 38, pp. 243–252. John Von Neumann Institute for Computing, Julich (2007)
15. Yu, J., Buyya, R.: A Taxonomy of Workflow Management Systems for Grid Computing. *Journal of Grid Computing* 3(3-4), 171–200 (2005)
16. Aldinucci, M., Danelutto, M.: Algorithmic Skeletons Meeting Grids. *Parallel Computing* 32(7), 449–462 (2006)
17. Aldinucci, M., Danelutto, M.: Securing skeletal systems with limited performance penalty: the Muskel experience. *Journal of Systems Architecture* 54(9), 868–876 (2008)
18. Bertolli, C., Coppola, M., Zocco, C.: The Co-replication Methodology and its Application to Structured Parallel Programs. In: CompFrame 2007: Proc. of the 2007 symposium on Component and framework technology in high-performance and scientific computing, pp. 39–48. ACM Press, New York (2007)
19. Danelutto, M., Zoppi, G.: Behavioural skeletons meeting services. In: Bubak, M., van Albdala, G.D., Dongarra, J., Sloot, P.M.A. (eds.) ICCS 2008, Part I. LNCS, vol. 5101, pp. 146–153. Springer, Heidelberg (2008)
20. Alves, A., et al.: Web Services Business Process Execution Language Version 2.0 (oasis standard). Technical report (2006)
21. Aldinucci, M., Bouziane, H., Danelutto, M., Pérez, C.: Towards a Spatio-Temporal sKeleton Model Implementation on top of SCA. Technical Report 0171, CoreGRID Network of Excellence (2008)
22. Apache Software Found: Tuscany home page, WEB (2008),
<http://tuscany.apache.org/>
23. Aldinucci, M., Danelutto, M., Zoppi, G., Kilpatrick, P.: Advances in autonomic components & services. In: Priol, T., Vanneschi, M. (eds.) From Grids To Service and Pervasive Computing (Proc. of the CoreGRID Symposium 2008), CoreGRID, Las Palmas, Spain, pp. 3–18. Springer, Heidelberg (2008)

Grid-Enabling SPMD Applications through Hierarchical Partitioning and a Component-Based Runtime

Elton Mathias, Vincent Cavé, Stéphane Lanteri, and Françoise Baude

INRIA Sophia-Antipolis, CNRS, I3S, UNSA

2004, Route des Lucioles, BP 93, F-06902 Sophia-Antipolis Cedex, France

{Elton.Mathias, Vincent.Cave, Stephane.Lanteri, Françoise.Baude}@inria.fr

Abstract. Developing highly communicating scientific applications capable of efficiently use computational grids is not a trivial task. Ideally, these applications should consider grid topology 1) during the mesh partitioning, to balance workload among heterogeneous resources and exploit physical neighborhood, and 2) in communications, to lower the impact of latency and reduced bandwidth. Besides, this should not be a complex matter in end-users applications. These are the central concerns of the DiscoGrid project, which promotes the concept of a hierarchical SPMD programming model, along with a grid-aware multi-level mesh partitioning to enable the treatment of grid issues by the underlying runtime, in a seamless way for programmers. In this paper, we present the DiscoGrid project and the work around the GCM/ProActive-based implementation of the DiscoGrid Runtime. Experiments with a non-trivial computational electromagnetics application show that the component-based approach offers a flexible and efficient support and that the proposed programming model can ease the development of such applications.

Keywords: Grid Computing, HPC, SPMD, Numerical Simulation, Software Components, GCM.

1 Introduction

Computational grids have proved to be particularly well suited to compute intensive embarrassingly parallel applications. The situation is less clear for parallel scientific applications, such as simulations involving the numerical resolution of partial differential equations (PDEs) using domain-decomposition techniques.

From the scientific application perspective, to develop large simulation software capable to adapt and evolve with the executing environment is a difficult task. Ideally, grid applications should consider network topology and heterogeneity of resources to preserve performance by balancing processing load and reducing communication through slower links.

This paper first introduces the DiscoGrid Project (Sec. 2) that addresses grid issues by promoting a new paradigm for grid-enabling parallel scientific applications to distributed and heterogeneous computing platforms. After, we

present a GCM/ProActive-Based implementation of the DiscoGrid Runtime (Sec. 3). Then, we present related works (Sec. 4). We also present experiments (Sec. 5) that were carried with a non-trivial computational electromagnetics application.

2 The DiscoGrid Project

The general objective of the DiscoGrid project, funded by the French National Research Agency (ANR), is to study and promote a hierarchical SPMD (Single Program, Multiple Data) programming model for the development of non-embarrassingly parallel scientific applications on computational grids, that can be considered inherently hierarchical and heterogeneous. In practice, the idea is to enable developers to design and implement applications, letting the DiscoGrid framework take care of grid related issues. The approach to achieve this objective is to improve both the conception and execution of parallel algorithms.

The conception of a parallel algorithm can be divided in three main parts: (i) data handling, (ii) communication and, obviously, (iii) the numerical computation. DiscoGrid includes an API to handle input data and the communication process is simplified through a set of advanced collective communication primitives (Sec. B.2), conceived to design and implement domain-decomposition based application or *gridify* existing ones.

DiscoGrid solves grid requirements by featuring a multi-level grid-aware partitioner (Sec. 2.2) and the support to deployment and communication is offered by a runtime (Sec. 3) that follows the DiscoGrid specification. The next subsections give further details about target applications, the partitioner, the hierarchical SPMD paradigm and the DiscoGrid API specification.

2.1 Target Applications

The DiscoGrid project focus on finite element simulation software such as those used for computational electromagnetism (CEM) and computational fluid flow problems (CFD) applications, but results can be applicable to other applications, parallelized upon a domain decomposition approach. CEM and CFD are nowadays involved in numerous industrial applications such as antenna design, electronic devices electromagnetic compatibility, aerodynamic, etc.

The traditional way of designing these simulations is to adopt an SPMD technique that combines mesh partitioning with the message passing programming model. The heterogeneity in network and resources is a particularly challenging issue for these applications because they involve a bulk synchronous approach where applications run at the pace of the slowest process.

Therefore, the hierarchical network topology and heterogeneity must impact the design of numerical algorithms that are both compute- and data-intensive. In addition, applications must be able to adapt to resources availability in order to avoid unbalances that may reduce the overall simulations performance.

2.2 Grid-Aware Multi-level Mesh Partitioning

The DiscoGrid multi-level partitioner is responsible to partition a finite element mesh taking into consideration available resources to balance the processors load and reduce inter-cluster communications, reasonably assuming that intra-cluster communications outperform inter-cluster ones. The partitioner interacts with the ParMetis [14] partitioner by providing iteratively the fractions of elements to be allocated to each partition, depending on the number of levels.

The runtime provides resources information (number and characteristics of machines, topology and network characteristics) and the *Partitioning* process (Fig. 1) takes care of converting the mesh into a graph and partitioning it. The *Renumbering* module takes care of renumbering nodes of the submeshes. Then, the *Submeshes creation* module builds the submeshes which can be partitioned again, depending on the defined number of levels. The output consists on a set of submeshes and partitioning information.

Empirical experiments comparing the multi-level partitioner with a one-level partitioner shown that the amount of inter-cluster communication can be reduced by 3 in the best case while increasing just by 3-5% the amount of data shared on interfaces in the worst case [13].

2.3 Hierarchical SPMD Programming Paradigm

The concept of hierarchical SPMD programming is in the core of the communication process of DiscoGrid applications. This paradigm aims at dealing with differences on LAN/WAN performances by organizing hierarchically the communications according to the multi-level partitioning described in the Sec. 2.2.

Fig. 2 shows an example of a two-level non-overlapping partitioning of a triangular mesh between two different clusters, with 4 and 9 nodes, respectively. Being Ω the global domain, the following entities can be identified: the subdomains Ω_i resulting from the higher level partitioning with interfaces between neighboring subdomains Ω_i and Ω_j denoted $\Gamma_{i,j}$; $\Omega_{i,k}$ are the subdomains resulting from the lower level partitioning. In the latter case, we shall make the distinction between two types of interfaces: $\beta_{i,k}^{i,l}$ denotes an interface between two neighboring subdomains $\Omega_{i,k}$ and $\Omega_{i,l}$ that belong to the same first level

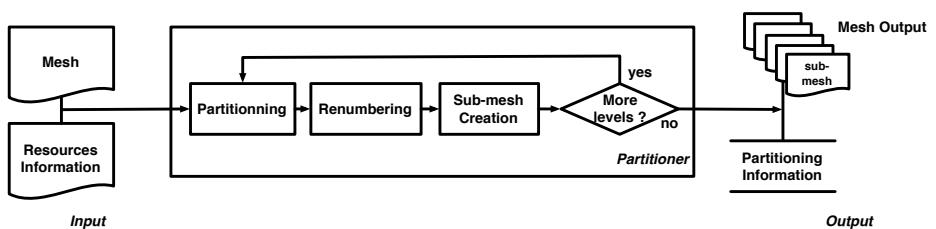


Fig. 1. Multi-level Mesh Partitioner Operation

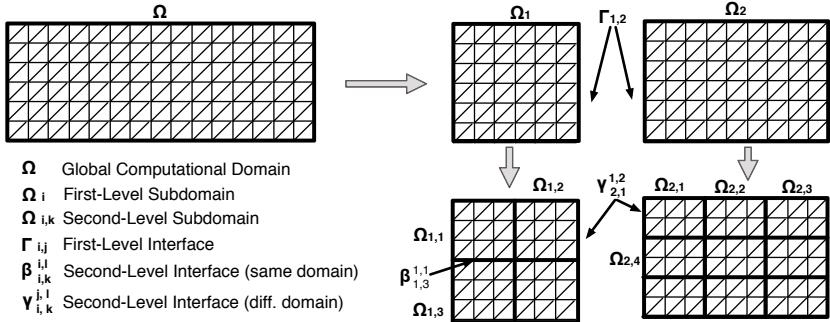


Fig. 2. Multi-level Partitioning

subdomain Ω_i ; $\gamma_{i,k}^{j,l}$ denotes an interface between two neighboring subdomains $\Omega_{i,k}$ and $\Omega_{j,l}$ that belong to different subdomains Ω_i and Ω_j .

Interfaces at each level of the partitioning (Γ , β and γ) have different communication requirements, depending on the network connecting (or not) the different processes. The hierarchical SPMD programming paradigm relies on this distinction of the different interfaces to optimize communications, for instance using high performance networks (Myrinet, SCI, etc.) whenever possible, staging and parallelizing collective communications and so on. These optimizations are not defined in the specification but achieved by the different runtime implementations, according to the underlying technology. Sec. 3 details the runtime that implements this model.

2.4 The DiscoGrid API

The *DiscoGrid API* [12], offered in C/C++ and Fortran, aims at providing programmers with primitives adapted to the development of mesh based PDE solvers. The originality of the DiscoGrid communication API resides in offering a way to develop hierarchical algorithms by using *hierarchical identifiers* to involve and target different sets of processes, possibly placed on different clusters/sites.

The DiscoGrid communication API defines classical point-to-point primitives such as *Send* and *Receive* as well as collectives ones such as *Barrier*, *Reduce*, *AllReduce*. The API also defines the *update* primitive, which is an automated way of exchanging distributed data shared in domain borders, using an associated meta-information. In DiscoGrid applications, the *update* primitive is used to exchange data at interfaces of each domain according to the metadata provided by the partitioner (Sec. 2.2) at deployment time.

The following Fortran subroutine, extracted from the CEM application (Sec. 5), is responsible for exchanging subdomain borders. The developer just needs to make use of the variable `dataname`, provided by the partitioner, to inform the DiscoGrid Runtime of the data to be communicated while `datadesc` holds all the information concerning the processes involved in the communication.

```

SUBROUTINE EXCHG1U(nffra, nutfro, vlfaca, vljt, dataname, datadesc)

INTEGER nffra
INTEGER nutfro(nfrmax)
REAL*8 vlfaca(nfrmmax), vljt(ntmax)
CHARACTER*6 dataname
INTEGER datadesc
(... data initialization ... )
CALL DG_IUPDATE(vlfaca, vljt, dataname, datadesc, dg ierr)
RETURN
END

```

3 The GCM/ProActive-Based DiscoGrid Runtime

The DiscoGrid Runtime is a component-based infrastructure that models a hierarchy of resources and offers support to the concepts and programming primitives described in Sec. 2.4. It is composed of three main modules: the *Information Manager*, which offers information about topology and status of the application and environment; the *Data Manager*, which gives access to application data and associated meta-data and the *Communication Manager*, which is responsible for the selection of the most appropriate communication layer: either native MPI, that offers access to high performance networks and optimizations within a cluster, or ProActive that offers inter-cluster communication, possibly tunneled through the support of any transport protocol (RMISSH, HTTP, HTTPS, etc).

The next paragraphs present some background information about the ProActive middleware and the GCM/ProActive framework, the architecture of the component-based runtime and the communication process.

The ProActive Grid Middleware. [1] is a Java middleware which aims to achieve seamless programming for concurrent, parallel and distributed computing. It offers an uniform active object programming model, and these objects are remotely accessible via asynchronous method invocation. ProActive features an abstract descriptor-based deployment model and framework that provide users with the capability to deploy an application using numerous network protocols, cluster resource managers and grid tools, without changing the source code. The definition of the deployment also encompasses security, tunneling of communications, fault tolerance and support of portable file transfer operations. The GCM/ProActive Runtime takes profit of the ProActive descriptor-based deployment framework to transfer submeshes to computing nodes and eventually the required software and binaries, so simplifying the deployment process. ProActive also features an MPI code wrapping mechanism, adding capabilities for deployment and life-cycle control of MPI processes. ProActiveMPI also enables "MPI to/from Java" communications, through a set of MPI-like C functions and Java classes. Thus, it adds the capability for MPI processes potentially located in different domains, under private IPs or behind firewall to communicate via ProActive.

The Grid Component model (GCM). defines a component model for grids, taking the Fractal component model [4] as a basis for its specification. In the type system of the Fractal model, a component interface is defined by its name, its signature, its role (client or server), its contingency (mandatory or not) and its

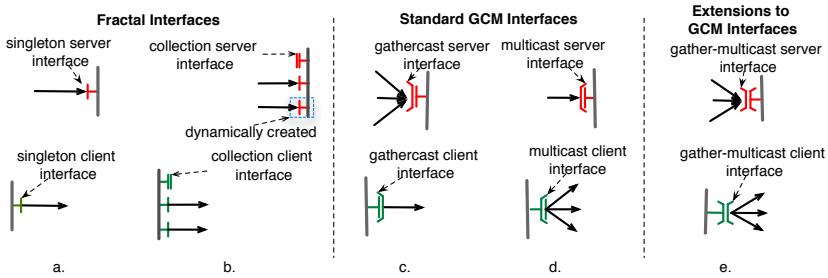


Fig. 3. GCM interfaces

cardinality. The Fractal model only defines two kinds of cardinalities: *singleton* (Fig. 3a) and *collection* (Fig. 3b). As these cardinalities only allow one-to-one communications, the GCM extends the Fractal model, introducing new *collective* cardinalities, namely *multicast* (Fig. 3c) and *gathercast* (Fig. 3d). These interfaces give the possibility to manage multiple interfaces as a single entity, and so expose the collective nature of components. In [9], we presented the design, implementation and evaluation of a mechanism to enable direct $M \times N$ communication through *gather-multicast* interfaces (Fig. 3e), integrated in the GCM/ProActive reference implementation of GCM.

3.1 The Component-Based Infrastructure

Fig. 4 shows the schematic composition of the DiscoGrid Runtime, considering the partitioning given by Fig. 2. It is a composition of two sorts of components:

- the **primitive wrapper**, depicted by the small components on Fig. 4, is the most elementary component used to form the entire component infrastructure. It wraps an MPI process and is responsible for the MPI to/from Java communication. By default, it only presents a single server interface and a single client interface that are bound to a generic **clustering** component that represents the upper level of the hierarchy. The primitive wrapper is also responsible for the encapsulation of MPI messages into objects with information useful to route messages through the component infrastructure. In Fig. 4 we can also notice that **primitive wrappers** are identified by a double-identifier (corresponding to a 2 level partitioning), which is identical to the subdomain identification ($\Omega_{j,l}$) assigned to the processing node. This identifier can be used to address a given process in the hierarchy.
- the **composite clustering** depicted on Fig. 4 by the bigger enclosing components are generic components capable of clustering lower-level components (**primitive wrappers** and **composite clustering** themselves) and serves as a gateway to incoming and outgoing messages. It presents two interfaces: a **gather-multicast server interface**, capable of receiving messages from other clustering components and dispatching messages to inner components and a **gather-multicast client interface** capable of gathering messages from inner-components and dispatch them to other clustering components of the same

level. These gather-multicast interfaces implement the main collective operations (reduce, scatter, gather, broadcast and barriers) as aggregation and dispatch policies, selected dynamically according to the requested operation.

MPI processes placed on a same cluster are able to communicate with each others through standard MPI. However, when the communication happens between different clusters, the communication is handled by the **wrapper** component. The **wrapper** components are bound to the enclosing **clustering** component that serves as a gateway to the given cluster. The **clustering** components on their turn are bound to each other through a gather-multicast interface that is capable to process messages when needed and route to the proper recipient, or group of recipients. By default, the entire component model is enclosed and bound to an external **clustering** composite, that exposes externally the entire DiscoGrid application. By doing this, we can envisage to compose new applications, e.g. loosely coupling simpler applications through GCM bindings or GCM component-web services.

3.2 High-Level DiscoGrid Communication Support

The *Communication Manager* of the DiscoGrid Runtime is in charge of translating DiscoGrid communications into one or many MPI or ProActive communications, depending on the operation type, source(s) and target(s). Next paragraphs explain each of the communication schemes.

Point-to-Point. A DiscoGrid point-to-point communication can either involve processes in the same cluster or not. In the former case, messages are translated into plain MPI communications whilst in the latter, a message is tagged in the wrapper component, sent upwards in the hierarchy and routed through gather-multicast interfaces until it reaches the destination. Worthy to note that the GCM implementation can rely upon a shortcut optimization called *tensioning*: the first call follows the complete path between sender and receiver and verify the connectivity between these components to create a shortcut that will be used afterwards to bypass composite components.

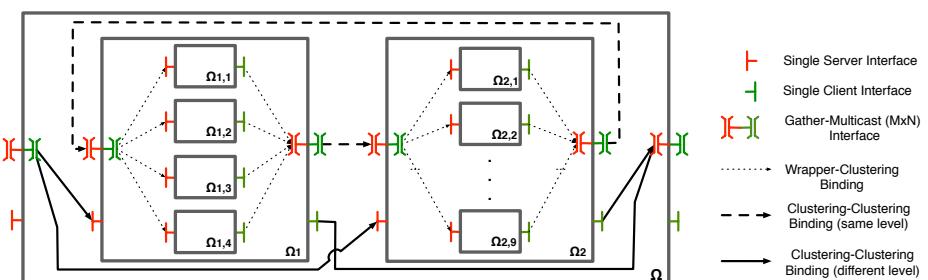


Fig. 4. Typical GCM-Based DiscoGrid Runtime Composition

Collective. As in MPI, DiscoGrid collective communications assume that all processes of a given level must participate. So, is said to take place at a given level. In general, messages are gathered on the most external clustering component that represents the given level of the hierarchy. Depending on the operation, the data is reduced on the local cluster before performing inter-cluster operations. Operations like *AllReduce* and *Barriers* require two phases, having the root process as the coordinator: first the runtime gathers or reduces data and after it dispatches the results or acknowledgments. Relying on GCM-based support, each phase is naturally parallelized.

Neighborhood-Based. This kind of communication is illustrated by the ‘update’ primitive. An update of the domain interfaces shared at a given level can be considered as a generalization of the MxN communication, that instead of just triggering a one shot communication between two sets of components (of size M and N), may spawn more than one MxN communications synchronizing all the borders of a given subdomain. Thanks to gather-multicast interfaces, this generalized MxN communication can be done at once: each process participates with its shared data of the given level, messages are then hierarchically gathered in the most external clustering component of the level, split and properly dispatched to the correct destinations, taking into account neighborhood information (Sec. 2.4) to split and determine who receives each piece of data.

3.3 Context-Based Communication Adaptations

Gather-multicast interfaces are a very general mechanism capable of performing any kind of communication, supposing one open channel between clustering components of the same level (even if it must be tunneled through SSH or use HTTP). In terms of performance, this cannot be considered an optimal solution: one or two synchronization barriers for each level involved on the call are required. Besides, the memory needed to aggregate messages at interfaces might be considerable. However, operations like Broadcasts, All-to-All, Reduction and Barriers may take profit of the gather-multicast interface to stage and parallelize their execution. On the contrary, efficient point-to-point operations require some form of direct communication between source and destination.

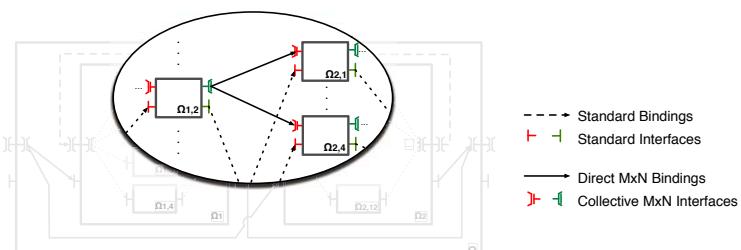


Fig. 5. Direct MxN Connection

Thanks to the component encapsulation, we are able to transparently configure direct bindings among **wrapper components** if a direct link between each pair of nodes is possible. The Fig. 5 shows the bindings resulting from the optimization of the update operation for the interfaces $\gamma_{2,1}^{1,2}$ and $\gamma_{2,4}^{1,2}$ of the Fig. 2. The general direct binding mechanism is detailed in [9].

4 Related Works

The support of SPMD applications on grid environments is mainly offered by two different approaches: either through grid-oriented implementations and extensions to the MPI standard or through a software layer that works as a glue, enabling loose coupling of SPMD codes.

Most of the tools that follow the first approach provide a transparent flat MPI environment focusing on optimizations on the communication layer. This is the case of MPICH-G2 [8], which develops a communication layer and infrastructure based on Globus. PACX-MPI [2] and GridMPI [10] offer optimizations, namely for efficient collective communications and support to heterogeneous networks.

The second approach focuses on loosely code coupling. This is the case of the efforts done in the context of the CCA MxN working group, like DCA [7], a distributed CCA framework based on an MPI-like interface and MxN data distribution operation and MetaChaos [6], a library that enables the definition of complex distributed and parallel data objects. These works suppose a small number of coupled components already internally structured, for instance as a standalone SPMD program, deployed on a single cluster.

Our approach can be considered an alternative to the first approach, leveraging the second one. We first offer abstractions to simplify the application programming, but we also offer a partitioner and runtime that cooperate to adapt the application to the underlying infrastructure at deployment time. Another advantage is the possibility to improve applications performances in a transparent way thanks to communication optimization offered by the GCM-based runtime. Besides, even with a foreseeable evolution of computing infrastructures towards cloud computing environments, applications could remain unchanged.

Regarding the multi-level partitioning, our approach is mainly based on the notion of hierarchical mesh partitioning, partially exploited in the context of the MecaGrid project [11]. Related works introduced the idea of autonomic application-sensitive partitioning along with adaptive mesh refinement [5] to handle space-time heterogeneity in runtime. Our current approach is simpler since the partitioning is only done before the execution, but our component-based runtime could also be very relevant in conjunction with such autonomic partitioning: decision of the partitioner to modify data placement at runtime, and so neighbourhood relationships, would naturally translate in modifying bindings among components, and so in a transparent manner for the end-user program.

5 Evaluation

As a non-trivial application for evaluating the DiscoGrid model, we consider a numerical modeling tool for the simulation of electromagnetic wave propagation in three-space dimensions. This tool is based on a finite element method working on arbitrarily unstructured tetrahedral meshes for solving the system of Maxwell equations. For the discretization of the system we adopt a discontinuous finite element method, also known as the discontinuous Galerkin method. From the computational point of view, the execution is characterized by two types of operations: purely local operations on tetrahedra for computing integral values and a calculation involving neighbor subdomains which involves a *gather-compute-scatter* sequence. Formulations are described in more details in [3].

Thanks to the DiscoGrid API, this sequence is depicted through an *update* operation that exchanges subdomain interfaces and local computations. After, the processing nodes participate in an AllReduce operation that is responsible for the calculation of the overall error rate, that defines the stop criteria.

5.1 Problem Setting and Experimental Environment

The problem considered here is the simulation of the scattering of the plane wave by an aircraft geometry. The computational domain is then defined as the volume between the aircraft and an artificial external boundary (a parallelepipedic box). The unstructured tetrahedral mesh which discretizes this volume consists of 225,326 vertexes and 1,305,902 tetrahedra (around 3GB of data).

These experiments were conducted in the Grid5000 testbed, on 8 different clusters spread all around France (in Sophia Antipolis, Lyon, Toulouse and Rennes), composed by machines of different processor architectures (Intel Xeon EM64T 3GHz, AMD Opterons 248 and 285) and memory from 2GB to 8GB by node. Cluster nodes are connected by gigabit ethernet switches, except 2 of them that are connected with Myrinet-10G and Myrinet-2000. The interconnection among the different clusters is done through a dedicated backbone with 2,5Gbit/s links. The implementation and evaluation of the DiscoGrid GCM/ProActive-based runtime uses Java Sun SDK v1.6.0_07 and ProActive v3.91. The MPI versions used on the experiments were MPICH v1.2.7p1 and GridMPI v2.1.1.

5.2 Performance Comparison

For comparison purposes, we benchmarked 3 different scenarios with the same application: the first using MPICH, the second using GridMPI and the third using a *gridified* implementation with the DiscoGrid API/Runtime. All the three scenarios make use of the DiscoGrid partitioner to handle resource heterogeneity but just the *gridified* version takes profit of the multi-level partitioning. Fig. 6 shows the CEM global execution time on 2, 4 and 8 clusters. Results show that GridMPI and DiscoGrid are better adapted to grid execution as they are around 10% faster than MPICH on 2 clusters and from 30% to 50% faster on 8 clusters.

Comparing DiscoGrid and GridMPI (Fig. 6), we can notice that GridMPI is, in general, faster than the DiscoGrid for small numbers of resources. This comes

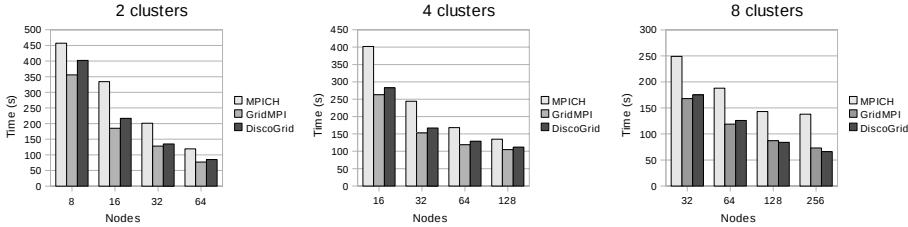


Fig. 6. CEM Execution Time for 2, 4 and 8 clusters

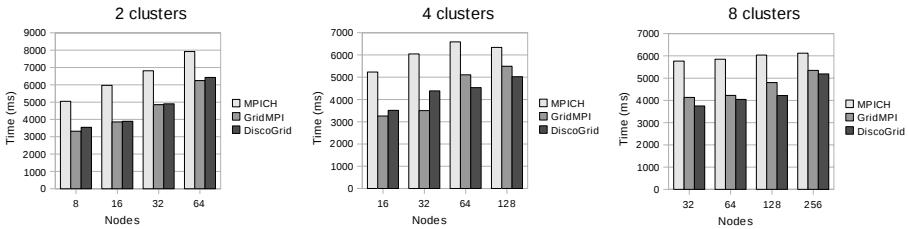


Fig. 7. Average Broadcast Time for 2, 4 and 8 clusters

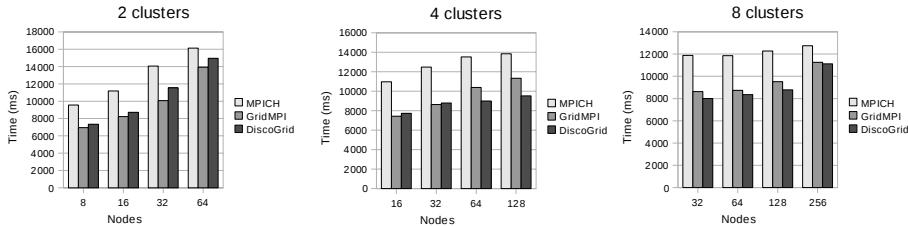


Fig. 8. Average AllReduce Time for 2, 4 and 8 clusters

from the fact that in such cases the amount of data communicated by each of the processes is bigger and the native-Java communication slows down the communication. Differently, with more clusters and nodes, the hierarchical and parallel treatment of communications overcame the cost of native-Java communication.

In order to verify in more details the performance of collective communications, the Fig. 7 presents the average time of a global Broadcast and the Fig. 8 brings the times for the AllReduce. Results have shown an improvement of collective communications performance starting from 64 nodes on 4 clusters and with 8 clusters. This partially explains the overall application execution time, as in such cases the DiscoGrid Runtime performed better than GridMPI due to a bigger number of nodes and the parallel execution of collective communication.

Even if for a small number of nodes, GridMPI is faster, we must consider that the real scenario of parallel simulations usually includes a large set of resources. In addition, the different benchmarks have shown that the DiscoGrid Runtime

is scalable, as the relative performance is improved when more clusters are used. Moreover, Grid5000 can be considered a fairly homogeneous platform and a more heterogeneous environment could take more profit of the grid-aware partitioner.

6 Conclusion and Perspective

In this paper, we presented some of the main concepts of DiscoGrid project and the work around a GCM/ProActive-based implementation of the DiscoGrid Runtime. This runtime supports the DiscoGrid API with emphasis on the performance of collective communication. It takes profit of extensions to the GCM to enable MxN communication on the context of grid environments, that can be considered heterogeneous and hierarchically organized.

Performance results obtained with only a few hundred nodes are promising. In particular, Broadcast and AllReduce performances seem to scale well. Thus, we can expect the usage of the Discogrid framework to improve significantly the overall applications performance. Besides, the usage of the DiscoGrid API tends to simplify the development of domain-decomposition based grid applications, as grid issues and data management are handled by the runtime and partitioner.

Our current work is to optimize native-Java communications to improve overall DiscoGrid performance. In addition, we intend to evaluate DiscoGrid on more heterogeneous environments, expecting taking further advantage of the hierarchical SPMD programming model and GCM features. Targeted environments are a combination of resources from clusters, production grids and clouds.

We also expect to use the DiscoGrid component-based infrastructure, namely the gather-multicast interfaces and the context-based MxN optimization to different application domains, e.g. to act as a message routing middleware to support an Internet-wide federation of enterprise service buses.

Acknowledgments. This work is partly carried out under the DiscoGrid project funded by the French ANR (Contract ANR-05-CIGC-005) under the framework of the program Calcul Intensif et Grilles de Calcul.

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, an initiative from the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners (see <https://www.grid5000.fr>)

References

1. Baduel, L., Baude, F., Caromel, D., Contes, A., Huet, F., Morel, M., Quilici, R.: Programming, Deploying, Composing, for the Grid. In: Grid Computing: Software Environments and Tools. Springer, Heidelberg (2006)
2. Beisel, Gabriel, Resch: An extension to MPI for distributed computing on MPPs. In: PVM/MPI, pp. 75–82 (1997)
3. Bernacki, M., Lanteri, S., Piperno, S.: Time-domain parallel simulation of heterogeneous wave propagation on unstructured grids using explicit non-diffusive, discontinuous Galerkin methods. J. Comp. Acoustics 14(1), 57–81 (2006)

4. Bruneton, E., Coupage, T., Stefani, J.B.: The fractal component model specification (2004), <http://fractal.objectweb.org/specification/index.html>
5. Chandra, S., Parashar, M.: Towards autonomic application-sensitive partitioning for samr applications. *J. Paral. Distrib. Comput.* 65(4), 519–531 (2005)
6. Edjlali, G., Sussman, A., Saltz, J.: Interoperability of data parallel runtime libraries. In: Proceedings of the Eleventh International Parallel Processing Symposium. IEEE Computer Society Press, Los Alamitos (1997)
7. Gannon, D., Krishnan, S., Fang, L., Kandaswamy, G., Simmhan, Y., Slominski, A.: On building parallel & grid applications: Component technology and distributed services. *Cluster Computing* 8(4), 271–277 (2005)
8. Karonis, N., Toonen, B., Foster, I.: MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing (JPDC)* 63(5), 551–563 (2003)
9. Mathias, E., Baude, F., Cavé, V.: A GCM-Based Runtime Support for Parallel Grid Applications. In: Proceedings of the Workshop on Component-Based High Performance Computing (CBHPC 2008) (October 2008)
10. Matsuda, M., Kudoh, T., Ishikawa, Y.: Evaluation of mpi implementations on grid-connected clusters using an emulated wan environment. In: Proceedings CCGRID 2003, Washington, DC, USA, p. 10. IEEE Computer Society, Los Alamitos (2003)
11. Mesri, Y., Digonnet, H., Guillard, H.: Mesh partitioning for parallel computational fluid dynamics applications on a grid. *Finite Vol. for Complex App.* (2005)
12. DiscoGrid Project. Spec. of an API for Hierar. Comm. Tech. report, INRIA (2007)
13. The DiscoGrid Project. Multi-level partitioning tool. Tech. report, INRIA (2008)
14. Schloegel, K., Karypis, G., Kumar, V.: Parallel static and dynamic multi-constraint graph partitioning. *Concurrency and Computation: Practice and Experience* 14(3), 219–240 (2002)

Reducing Rollbacks of Transactional Memory Using Ordered Shared Locks

Ken Mizuno, Takuya Nakaike, and Toshio Nakatani

Tokyo Research Laboratory, IBM Japan, Ltd.
{kmizuno, nakaike, nakatani}@jp.ibm.com

Abstract. Transactional Memory (TM) is a concurrency control mechanism that aims to simplify concurrent programming with reasonable scalability. Programmers can simply specify the code regions that access the shared data, and then a TM system executes them as transactions. However, programmers often need to modify the application logic to achieve high scalability on TM. If there is any variable that is frequently updated in many transactions, the program does not scale well on TM.

We propose an approach that uses ordered shared locks in TM systems to improve the scalability of such programs. The ordered shared locks allow multiple transactions to update a shared variable concurrently without causing rollbacks or blocking until other transactions finish. Our approach improves the scalability of TM by applying the ordered shared locks to variables that are frequently updated in many transactions, while being accessed only once in each transaction. We implemented our approach on a software TM (STM) system for Java. In our experiments, it improved the performance of an hsqldb benchmark by 157% on 8 threads and by 45% on 16 threads compared to the original STM system.

1 Introduction

Transactional Memory (TM)  is a new concurrency control mechanism that aims to simplify concurrent programming with acceptable scalability. In TM, program sections that access shared variables are treated as transactions. If some transactions cause a conflict, as when one transaction updates a variable that another concurrently executing transaction has read, then the TM system rolls back and re-executes one of the transactions.

If there is a variable that is updated in many transactions, it will cause frequent conflicts and degrade the scalability of the program. Figure  is a sample program for a hash table. Some transactions are tagged using the keyword `atomic`. In this program, `size` is updated every time an entry is inserted or removed, and this causes frequent conflicts. Figure  shows a sample execution sequence of two concurrent transactions in a TM system that uses lazy conflict detection . They read the same value from `size` and increment it. The update in each transaction is invisible to the other transaction before it is committed. If Transaction A commits successfully before Transaction B tries to commit, then B should roll back because `size` was updated by the commit of A after the read operation in B. In contrast, if `size` is eliminated, conflicts rarely occur since the accesses to `table` are scattered by the hashes of the keys. A similar situation

```

public void put(Object k, Object v) {
    atomic {
        if (contains(k)) {
            updateRecord(table[hash(k)], k, v);
        } else {
            int s = ++size;
            if (s > capacity) expandTable();
            insertRecord(table[hash(k)], k, v);
        }
    }
}

public void remove(Object k) {
    atomic {
        if (contains(k)) {
            size--;
            removeRecord(table[hash(k)], k);
        }
    }
}

public Object get(Object k) {
    atomic {
        return getRecord(table[hash(k)], k);
    }
}
}

```

Fig. 1. Sample program for a hash table with a counter variable

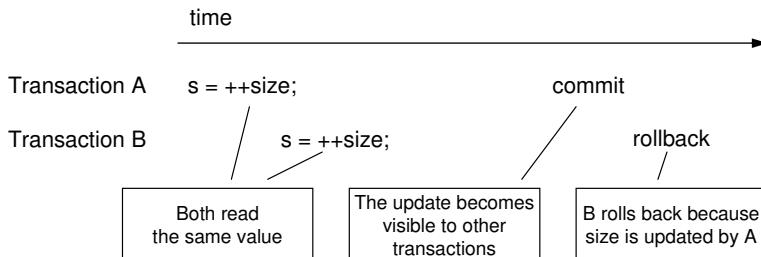


Fig. 2. Sample execution sequence of two transactions accessing hash table

occurs for the variables that hold the next ID numbers in ID number generators, variables that hold the sizes of collection objects, or variables that hold execution statistics. If a program is not tuned for TM, it often contains such variables and they prevent the scalability.

In this paper, we propose an approach for TM systems that uses ordered shared locks [4][5][6] for the variables that are tagged by the programmer. The ordered shared locks allow multiple transactions to update a shared variable concurrently without causing rollbacks or blocking until other transactions finish. If two transactions are executed in the same order as the example of Figure 2 using the ordered shared lock for `size`,

Transaction B reads the result of the increment operation in Transaction A, and both A and B successfully commit.

In this paper, we make the following contributions.

- We present an approach that uses ordered shared locks in TM.
- We explain an implementation of the approach on a software-based TM (STM) system for Java.
- We evaluate the performance of the approach using some benchmark programs.

We give a brief overview of the original ordered shared lock in Section 2. We explain our approach and an implementation of it as an STM system for Java in Section 3. We evaluated the performance of the implementation of our approach in Section 4. We review related work in Section 5 and conclude in Section 6.

2 Ordered Shared Lock

The ordered shared lock [4][5][6] is an extension of the shared/exclusive lock that allows multiple transactions to share locks even when one of the locks is a lock for write access.¹ Locks between multiple reads are shared as in the traditional shared/exclusive lock.

Figure 3 is a sample execution sequence of transactions that use ordered shared locks. Transaction B can acquire a lock on X even though Transaction A has acquired a lock on X. In this case, Transaction B observes the result of the write operation in A even though A has not committed yet. The lock on X is said to be *on hold* for Transaction B since it was acquired after Transaction A acquired the lock on X and before Transaction A releases the lock. The lock for B remains on hold until Transaction A releases the lock. When more than two transactions acquire locks for the same object concurrently, the lock for Transaction B is on hold until all of the transactions that acquired the lock before Transaction B acquired the lock release their locks.

Transactions that use ordered shared locks should satisfy the following constraints.

Acquisition Rule. If Transaction A acquires a lock on object X before Transaction B acquires a lock on object X, the operation on X in Transaction A must precede the operation on X in Transaction B.

Relinquishing Rule. A transaction may not release any lock as long as any of its locks are on hold.

The example in Figure 3 satisfies the constraints since Transaction B reads X after Transaction A wrote to X, and the release operation for the lock on X of Transaction B is deferred until A releases the lock on X.

The ordered shared lock may cause cascading rollbacks. If Transaction A is rolled back in the example of Figure 3, then Transaction B should also be rolled back because it has read the result of an operation from Transaction A. Therefore, the ordered shared lock is not suitable for transactions that cause rollbacks frequently because it causes frequent cascading rollbacks.

¹ We describe only the most permissive protocol of the ordered shared locks that allows ordered sharing both between read and write, and between write and write.

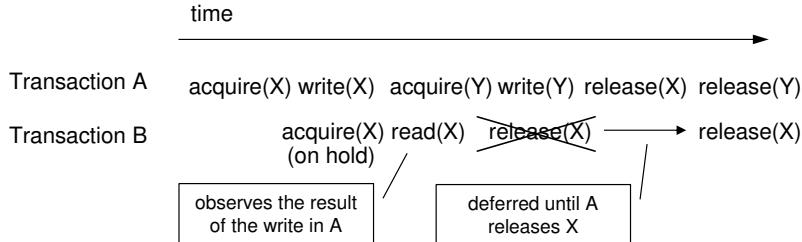


Fig. 3. Sample execution sequence of two transactions with ordered shared locks

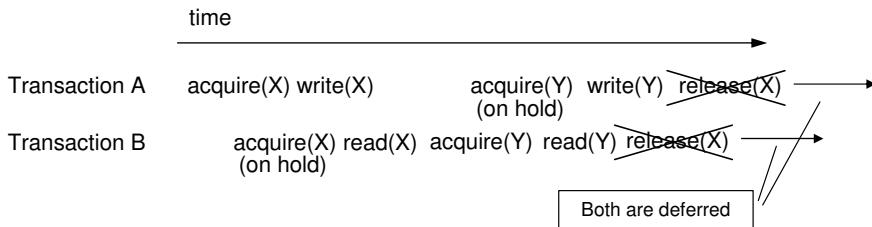


Fig. 4. Sample execution sequence that causes deadlock

The ordered shared lock approach may also cause a deadlock. However, deadlock can be detected and recovered from by using a traditional deadlock detection and recovery technique. A simple way to recover from a deadlock is to roll back one of the deadlocked transactions. Figure 4 is a sample execution sequence that causes a deadlock. In this example, both Transaction A and B are unable to commit before the other transaction releases its locks.

3 Our Approach

In this section we introduce our approach that uses the ordered shared locks for TM. We first explain how to integrate the ordered shared locks in the TM. After that, we describe the design of our STM system for Java, and then we explain the implementation of our approach.

3.1 Integrating Ordered Shared Locks in TM

In our approach, programmers specify some variables as *uncooperative variables*, and then the TM system uses ordered shared locks for them. This prevents the rollbacks due to conflicts on the uncooperative variables and improves the scalability.

Programmers should specify the variables with these properties as the uncooperative variables:

- They are frequently updated in many transactions.
- They are accessed at most once in each transaction.
- Each transaction accesses at most one uncooperative variable.

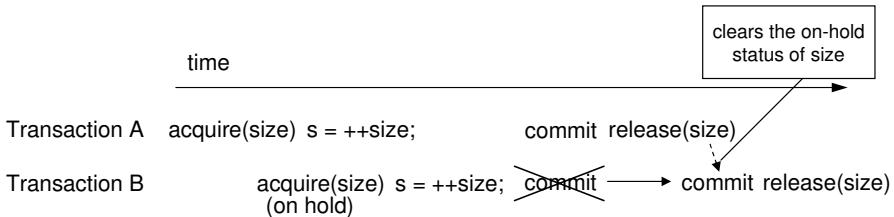


Fig. 5. Sample execution sequence for hash table using our approach

The variables that satisfy the first property cause frequent conflicts in the standard TM systems. The ordered shared locks are suitable for such variables because they prevent the rollbacks due to the conflicts. The other two properties avoid rollbacks. If one of these two properties does not hold, rollbacks can occur due to situations such as illustrated in Figure 4. We think these two properties are acceptable because our motivation is to remove the rollbacks caused by a few uncooperative variables.

In order to integrate the ordered shared locks into the TM, we divided the relinquishing rule described in Section 2 into two rules. Now the constraints of the ordered shared locks can be interpreted as:

Acquisition Rule. If Transaction A acquires a lock on object X before Transaction B acquires a lock on object X, then the operation on X in Transaction A must precede the operation on X in Transaction B.

Committing Rule. A transaction may not commit as long as any of its locks is on hold.

Relinquishing Rule. A transaction may not release any lock before it commits or aborts.

We say a transaction is *ready to commit* when it reaches the end of the transaction and none of its locks is on hold. The conflict detection for the variables other than the uncooperative variables is included in the commit operation when we are using the lazy conflict detection. It should be executed after the transaction becomes ready to commit, although it can be executed speculatively before the transaction becomes ready to commit to detect conflicts and abort more quickly. Figure 5 shows a sample execution sequence for the example in Figure 1 using our approach. Now the two transactions successfully commit and `size` is incremented twice.

A transaction with ordered shared locks will roll back in these situations:

- When a conflict occurs in variables that do not use ordered shared locks.
- When a deadlock is detected in the transaction.
- When a transaction that this transaction is waiting for is rolled back. (A cascading rollback.)

The first situation is the same as the original TM. The main benefit of the ordered shared lock is that the uncooperative variables do not cause any rollback in this situation. The second and the third situations are caused by the ordered shared lock.

Our approach is strongly serializable [7], which means that the transactions are serializable in the order of the commit operations, as long as it is applied to a TM system that is also strongly serializable. To prove this, we need to check this property: The

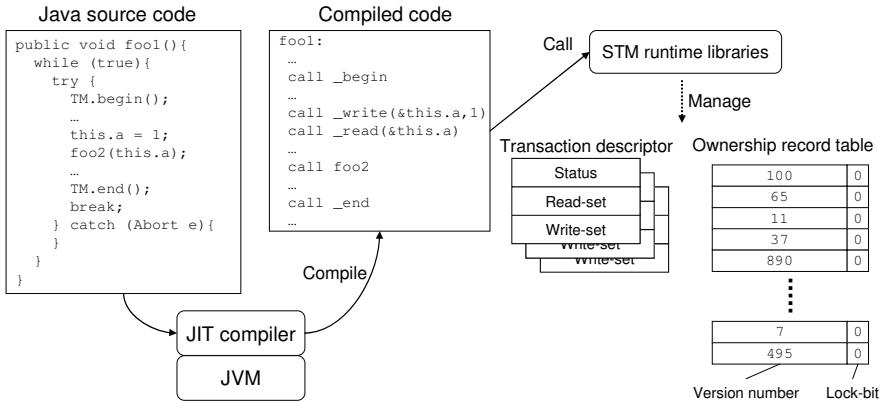


Fig. 6. Overview of Our STM System

commit operation of Transaction A must finish before the beginning of the commit operation of Transaction B if Transaction B reads the value of the variable that Transaction A wrote. If the variable is an uncooperative variable, this property is guaranteed by the three rules of this section. If the variable is not an uncooperative variable, this property is satisfied as long as the TM system is strongly serializable. Most TM systems that use lazy conflict detection guarantees this property by executing the validation and commit atomically.

3.2 Design of a Java STM

Figure 6 shows an overview of our Java STM system. It consists of an STM runtime library that is based on an IBM STM for C/C++ [8], and a JIT compiler that generates code that calls the runtime library.

The JIT compiler generates two versions of JIT-compiled code for each method invoked in a transaction: a non-transactional version and a transactional version [9][10]. In the transactional version, the `stm_read` or `stm_write` function of the runtime library is called when the code accesses the heap area. Transactions are specified using special methods `TM.begin()` and `TM.end()`.

We used lazy conflict detection and lazy versioning [9][11]. With lazy versioning, the results of the write operations in a transaction are buffered to a transaction-local data structure. The results are reflected to the shared variables when the transaction successfully commits, but they are discarded if the transaction rolls back. The alternative is eager versioning [10][12], which stores the result of a write operation directly in the shared variable, and copies the old values to a transaction-local data structure in order to write them back to the shared variable if the transaction rolls back.

To simplify the memory management of the STM runtime, we do not require the STM operations to be non-blocking, but instead we use locking to access the metadata [12]. We devised a simple mechanism to avoid starvation in which a long-running transaction is repeatedly aborted by short running transactions. When the number of

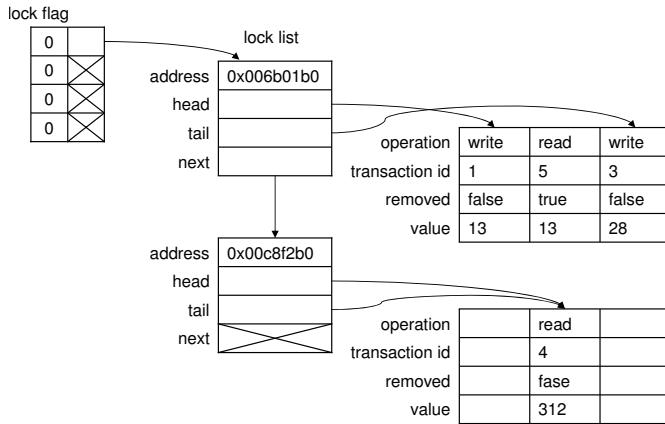


Fig. 7. Data Structure to Control Ordered Shared Lock

rollbacks of a transaction exceeds a threshold, the transaction acquires a lock to prevent the other transactions from proceeding to the commit process.

Our Java STM system provides weak atomicity, and thus it does not guarantee the safety of publication or privatization [11][13].

When garbage collection (GC) occurs, we abort all of the transactions. This is because GC may move objects and invalidate the metadata of the transactions. We use periodic validations to avoid infinite loops due to inconsistent reads. When an inconsistent read causes an exception such as a “divide by zero” exception, the TM system catches the exception and executes the usual cleanup operations in `TM.end()`. If the exception is caused by an inconsistent read, then the validation fails and the transaction rolls back.

3.3 Implementation of Our Approach

In this subsection we describe our implementation of the ordered shared lock in our Java STM system. This implementation is based on the description in [5][6].

Figure 7 shows the data structure used to control the ordered shared locks. The system provides a *lock list* for each variable that uses the ordered shared locks. The lock list is implemented using an array and the array contains transactions which hold the ordered shared lock in the order of lock acquisition. The list also contains the types of the operations (read or write), flags that indicate whether the entry was removed due to a rollback, and the values that the transactions read or wrote. A write operation stores the value in the lock entry and it is reflected into the shared variable in the commit operation. A read operation reads the value from the preceding lock entry if it exists. The lock lists are stored in a hash table using the addresses of the variables as keys. Chaining is used when a hash collision occurs.

Each record of the hash map has a *lock flag* field. In order to satisfy the acquisition rule, the lock acquisition and a read or write operation are executed atomically using the *lock flag*: the transaction sets the *lock flag*, appends an entry to the *lock list*, executes

the operation, and then clears the lock flag. We also implemented a method that adds a specified value to a specified variable atomically for use in increment or $+=$ operations. If such operations are executed as a combination of read and write operations, another transaction may access the variable in the middle of the operations. This may cause a deadlock.

In order to implement the commit rule, each transaction has a variable named `wait_count`, which indicates the number of locks that are on hold. The variable is incremented when the transaction acquires a lock that will be on hold. When a transaction releases a lock in a commit or rollback operation, it decrements the `wait_count` for the transaction whose lock released the on-hold status. When a transaction reaches its end, it waits until `wait_count` becomes zero before executing the validation. To detect a deadlock, the transaction will be rolled back if it times out before `wait_count` becomes zero. This approach has little overhead for the deadlock detection, but it causes a significant slowdown when a deadlock occurs. We chose this approach because the properties of the uncooperative variables described in Section 3.1 avoid deadlock and the deadlock detection mechanism is rarely triggered.

4 Experiments

This section describes the benchmark experiments that we watched to analyze the performance of our approach.

4.1 Benchmark Programs

We tested three programs: a small program intensively accessing `HashMap`, the SPEC-jbb2005 [14] benchmark, and the `hsqldb` benchmark [15] in the DaCapo benchmarks [16]. We executed them in the synchronized mode, the TM mode without ordered shared locks, and the TM mode with ordered shared locks. In the synchronized mode, each critical section is guarded using Java's synchronized block. In the TM modes, each critical section is executed as a transaction. We used 10-millisecond intervals for the periodic read-set validations in the TM modes. We executed the benchmarks on Red Hat Enterprise Linux Server release 5.1 on a 16-way POWER6 machine. Our STM system for Java is based on the IBM 64-bit JVM version 1.5.0. We used 8 GB of Java heap to minimize the GC events in our experiments.

HashMap. We implemented a benchmark program for `java.util.HashMap`. In our benchmark, a `HashMap` object is first filled with 1,000 elements and then each thread calls one of the `get`, `put`, or `remove` methods in a critical section. The critical section is guarded by synchronizing the `HashMap` object, but it is executed as a transaction in the TM modes. When a thread calls the `put` method, it always puts a new entry rather than updating an existing entry. The transactions are 95% calls to `get`, 2.5% calls to `put`, and 2.5% calls to `remove`.

For the TM mode with ordered shared locks, we specified as an uncooperative variable the variable named `elementCount`, which holds the number of elements and which is updated in the `put` and `remove` methods. In the `put` method, this variable is also used to check if it exceeds the threshold to resize the table. We set the initial size of the `HashMap` large enough to prevent any resize operations during an experimental run.

SPECjbb2005. We used a modified version of SPECjbb2005. The original SPECjbb-2005 prepares a thread-local data structure called a *warehouse* to avoid lock contentions. We modified it to share a single warehouse for all of the threads, similar to the implementation of Chung et al. [17]. Each SPECjbb2005 transaction (such as *new order* or *order status*) is executed in a critical section. However, the JBBDataStorage² transaction that traverses multiple tables is divided into multiple critical sections, each of which traverses one table. The critical sections are guarded by synchronizing the warehouse object in the synchronized mode.

For the TM mode with ordered shared locks, we specified as an uncooperative variable the variable named `historyCount`, which is always updated in a SPECjbb2005 transaction named `DeliveryTransaction`.

Hsqldb. We used the hsqldb benchmark in the DaCapo benchmarks. In this benchmark, the database tables are placed in memory, and an SQL statement is executed in a critical section. Hsqldb is not designed to execute SQL statements concurrently, and there is some shared data that does not actually need to be shared. We modified the code to avoid rollbacks caused by the shared data:

- There is a counter variable that tracks the number of created objects. It is used to trigger garbage collection at a specified threshold. We removed the counter since it provides little benefit in a multi-threaded environment.
- Some hash tables are used to store data that is essentially thread-local. This is implemented as shared data, which we modified to use thread-local hash tables.
- Hsqldb uses object pools to increase the single-thread performance. We disabled them because they decrease the multi-thread performance.

For the TM mode with ordered shared locks, we specified as an uncooperative variable the variable named `currValue` in the `NumberSequence` class. This variable is used to generate the ID numbers of the database records.

4.2 Result

Figure 8 shows the relative throughput of the benchmarks compared to the single-thread performance of the synchronized mode. Figure 9 shows the abort ratio, the ratio of aborted transactions to all of the transactions. The label “OS” denotes the ordered shared lock in the figures.

In all of the benchmarks, the results of the TM modes are worse than that of the synchronized mode when the number of threads is small. This is because of the overhead of the software-based TM.

In the `HashMap` benchmark, our approach shows a low abort ratio because we eliminated the conflicts on the uncooperative variable. However, the throughput was degraded because the path length of the commit operation is relatively long in the kernel loop of this benchmark. In our approach, the commit operations in transactions that access the same uncooperative variable are serialized because no transaction can start a commit operation until the preceding transactions release the ordered shared lock. As

² Implemented using `java.util.TreeMap`.

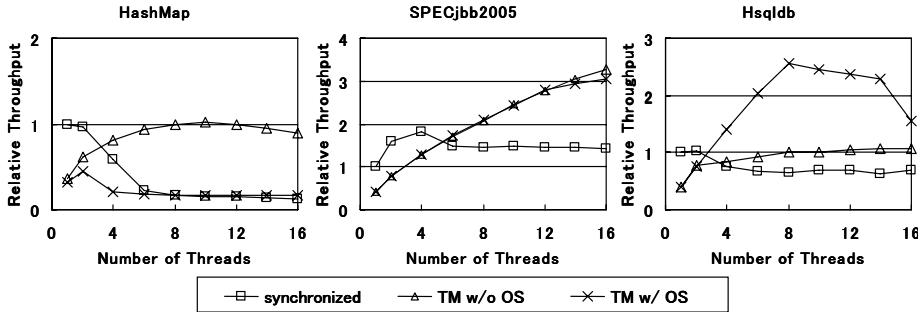


Fig. 8. Relative throughput compared to the single-thread performance of the synchronized mode (higher is better)

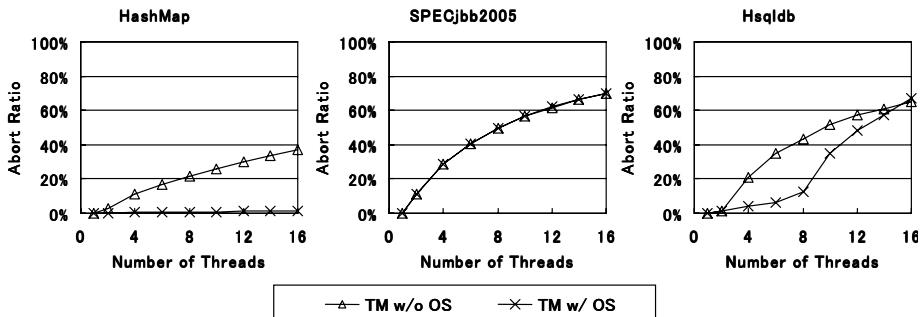


Fig. 9. Abort ratio (lower is better)

the number of rollbacks is reduced, it will be more beneficial when HashMap is used in larger transactions.

Our approach did not improve the performance of the modified SPECjbb2005 benchmark. In this benchmark, reducing the conflicts on one uncooperative variable does not reduce the abort ratio. There are other variables that cause conflicts but which are not suitable for the ordered shared lock.

Finally, the results for the hsqldb benchmark were greatly improved. The throughput increased by 157% on 8 threads and by 45% on 16 threads compared to the TM mode without ordered shared locks. The uncooperative variable was the primary cause of the conflicts, thus the ordered shared lock had provided a great benefit. On the down side, the abort ratio was increased when there were more than 8 threads. This seems to be caused by cascading rollbacks. When the abort ratio is very high, our approach causes further rollbacks due to the cascading rollbacks, which reduces the benefit of our approach.

5 Related Work

In order to reduce the costs of rollbacks, various kinds of nested transactions have been introduced. However, none of them are beneficial for the variables whose values affect

the execution of other parts of the transaction. For example, with closed nesting [18], the conflicts in an inner transaction are detected at the end of the inner transaction, so only the inner transaction is rolled back. Unfortunately, this provides no benefit when the conflicts occur between the end of the inner transaction and the end of the outer transaction. Open nesting [18] solves this problem, but the results of the inner transaction must not affect the other parts of the outer transaction. It is not applicable to transactions such as `put` in Figure 11 which use the results of increment operations. With abstract nesting [9], only an inner transaction is rolled back when conflicts caused by the inner transaction are detected at the end of the outer transaction. However, this provides no benefits if the result of the re-execution of the inner transaction affects other parts of the outer transaction.

The correctness of TM was discussed in [20]. This work introduces a correctness criterion called *opacity* that captures the inconsistency of reads. We did not use this approach since we can safely detect the exceptions caused by inconsistent reads in Java.

The ordered shared lock was proposed by Agrawal et al. [4, 5, 6]. The analysis of [5] showed that the ordered shared lock improves the performance for database workloads with large numbers of data contentions. The ordered shared lock has also been applied to real-time databases in [6] to eliminate blocking.

6 Conclusion

We proposed an approach that uses the ordered shared lock in TM systems. In this approach, frequently conflicting variables that are accessed only once in each transaction are controlled by ordered shared locks and the other variables are controlled by a normal conflict detection mechanism of the TM. This approach improves the scalability because we can reduce the conflicts caused by the uncooperative variables. In our experiments, we improved the performance of an hsqldb benchmark by 157% on 8 threads and by 45% on 16 threads compared to the original STM system.

References

1. Herlihy, M., Moss, J.E.B.: Transactional Memory: Architectural Support for Lock-Free Data Structures. In: 20th Annual International Symposium on Computer Architecture, pp. 289–300. ACM Press, New York (1993)
2. Hammond, L., Wong, V., Chen, M.K., Carlstrom, B.D., Davis, J.D., Hertzberg, B., Prabhu, M.K., Wijaya, H., Kozyrakis, C., Olukotun, K.: Transactional Memory Coherence and Consistency. In: 31st Annual International Symposium on Computer Architecture, pp. 102–113. ACM Press, New York (2004)
3. Fraser, K., Harris, T.: Concurrent Programming Without Locks. ACM Trans. Comput. Syst. 25(2), 1–61 (2007)
4. Agrawal, D., Abbadi, A.E.: Locks with Constrained Sharing. In: 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pp. 85–93. ACM Press, New York (1990)
5. Agrawal, D., Abbadi, A.E., Lang, A.E.: The Performance of Protocols Based on Locks with Ordered Sharing. IEEE Trans. Knowl. Data Eng. 6(5), 805–818 (1994)

6. Agrawal, D., Abbadi, A.E., Jeffers, R., Lin, L.: Ordered Shared Locks for Real-Time Databases. *VLDB J.* 4(1), 87–126 (1995)
7. Breitbart, Y., Garcia-Molina, H., Silberschatz, A.: Overview of Multidatabase Transaction Management. *VLDB J.* 1(2), 181–239 (1992)
8. XL C/C++ for Transactional Memory for AIX,
<http://www.alphaworks.ibm.com/tech/xlcstm/>
9. Harris, T., Fraser, K.: Language Support for Lightweight Transactions. In: 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, pp. 388–402. ACM Press, New York (2003)
10. Adl-Tabatabai, A.R., Lewis, B.T., Menon, V., Murphy, B.R., Saha, B., Shpeisman, T.: Compiler and Runtime Support for Efficient Software Transactional Memory. In: ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, pp. 26–37. ACM Press, New York (2006)
11. Shpeisman, T., Menon, V., Adl-Tabatabai, A.R., Balensiefer, S., Grossman, D., Hudson, R.L., Moore, K.F., Saha, B.: Enforcing Isolation and Ordering in STM. In: ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, pp. 78–88. ACM Press, New York (2007)
12. Saha, B., Adl-Tabatabai, A.-R., Hudson, R.L., Minh, C.C., Hertzberg, B.: McRT-STM: A High Performance Software Transactional Memory System for a Multi-Core Runtime. In: 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 187–197. ACM Press, New York (2006)
13. Menon, V., Balensiefer, S., Shpeisman, T., Adl-Tabatabai, A.R., Hudson, R.L., Saha, B., Welc, A.: Practical Weak-Atomicity Semantics for Java STM. In: 20th Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 314–325. ACM Press, New York (2008)
14. SPECjbb2005, <http://www.spec.org/jbb2005/>
15. hsqldb, <http://hsqldb.org/>
16. The DaCapo benchmark suite, <http://dacapobench.org/>
17. Chung, J., Minh, C.C., Carlstrom, B.D., Kozyrakis, C.: Parallelizing SPECjbb2000 with Transactional Memory. In: Workshop on Transactional Memory Workloads (2006)
18. Moss, J.E.B., Hosking, A.L.: Nested Transactional Memory: Model and Architecture Sketches. *Sci. Comput. Program.* 63(2), 186–201 (2006)
19. Harris, T., Stipic, S.: Abstract Nested Transactions. In: The 2nd ACM SIGPLAN Workshop on Transactional Computing, TRANSACT 2007 (2007)
20. Guerraoui, R., Kapalka, M.: On the Correctness of Transactional Memory. In: 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 175–184. ACM, New York (2008)

Topic 10

Parallel Numerical Algorithms

Introduction

Peter Arbenz*, Martin van Gijzen*, Patrick Amestoy*, and Pasqua D’Ambra*

Numerical algorithms have played a key role in parallel computing since its beginning. In fact, numerical routines have caused the highest demand for computing power anywhere, making their efficient parallelization one of the core methodical tasks in high-performance computing. Many of today’s fastest computers are mostly used for the solution of huge systems of equations as they arise in the simulation of complex large-scale problems in computational science and engineering.

Parallel numerics plays a fundamental role in the promotion of computational science and engineering. On the one hand, successful algorithms are further developed and adapted to exploit new hardware like multi-core architectures or graphics processing units. Massively parallel heterogeneous systems challenge the scalability of existing numerical schemes. On the other hand, new numerical techniques, like grid adaptation, mesh refinement, have been introduced that are difficult to parallelize, keeping loads balanced and communication reduced to a minimum. The papers that have been accepted for presentation at Euro-Par 2009 address these challenges in one way or the other.

Overall, 22 papers were submitted to this topic, with authors from Austria, Croatia, France, Germany, India, Italy, Japan, the Netherlands, Portugal, Russia, Spain, Switzerland, Turkey, and the United States. Out of these 22 submissions, nine were accepted for the conference. The outstanding paper “Wavelet based adaptive simulation of complex systems on multi-core architectures” by D. Rossinelli, M. Bergdorf, B. Hejazialhosseini, and P. Koumoutsakos has received a distinguished paper award. The paper deals with wavelet-based adaptive simulation of complex systems on multi-core architectures. Thus, it addresses two hot topics at once, how to handle adaptive discretizations in parallel and multi-core programming.

The eight regular papers are presented in three sections, one on *CFD and rigid body dynamics*, one on *discretisation (time and spatial)*, and one on *linear algebra*. This substructuring is reflected in the arrangement of the papers in the following part of the conference proceedings.

In the section on *CFD and rigid body dynamics*, Stefan Donath, Christian Feichtinger, Thomas Pohl, Jan Götz, and Ulrich Rüde improved their parallel Free Surface Lattice-Boltzmann method to simulate the coalescence of bubbles. Kathrin Burckhardt, Dominik Szczerba, Jed Brown, Krishnamurthy Muralidhar, and Gabor Székely devise an implicit Navier-Stokes solver to simulate the oscillatory flow in human abdomen. Klaus Iglberger, Ulrich Rüde present the

* Topic chairs.

first large scale rigid body dynamics algorithm. It is based on fast frictional dynamics.

In the section on *discretisation*, Werner Augustin, Vincent Heuveline, and Jan-Philipp Weiß investigate in-place and time-blocking techniques to optimize stencil computations on shared-memory multi-core machines. Matthias Korch and Thomas Rauber target the same type of machines for the memory-efficient solution of high-dimensional ODEs.

In the *linear algebra* section, Murat Manguoglu, Ahmed Sameh, and Olaf Schenk combine the sparse system solver Pardiso with the “Spike” banded system solver to obtain an efficient preconditioner for BiCGstab. Mercedes Marqués, Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, and Robert van de Geijn target the development of a high-performance QR factorization algorithm for dense matrix operations where data resides on disk and has to be explicitly moved in and out of the main memory. Fangbin Liu, and Frank J. Seinstra present an adaptive parallel Householder Bidiagonalization algorithm that they apply to multimedia content analysis.

Altogether, the contributions to Topic 10 at the 2009 Euro-Par in Delft show once more the great variety of interesting, challenging, and important issues in the field of parallel numerical algorithms. Thus, we are already looking forward to the new results submitted to and presented at next year’s Euro-Par conference.

Wavelet-Based Adaptive Solvers on Multi-core Architectures for the Simulation of Complex Systems

Diego Rossinelli, Michael Bergdorf, Babak Hejazialhosseini,
and Petros Koumoutsakos

Chair of Computational Science, ETH Zürich, CH-8092, Switzerland

Abstract. We build wavelet-based adaptive numerical methods for the simulation of advection dominated flows that develop multiple spatial scales, with an emphasis on fluid mechanics problems. Wavelet based adaptivity is inherently sequential and in this work we demonstrate that these numerical methods can be implemented in software that is capable of harnessing the capabilities of multi-core architectures while maintaining their computational efficiency. Recent designs in frameworks for multi-core software development allow us to rethink parallelism as task-based, where parallel tasks are specified and automatically mapped into physical threads. This way of exposing parallelism enables the parallelization of algorithms that were considered inherently sequential, such as wavelet-based adaptive simulations. In this paper we present a framework that combines wavelet-based adaptivity with the task-based parallelism. We demonstrate good scaling performance obtained by simulating diverse physical systems on different multi-core and SMP architectures using up to 16 cores.

1 Introduction

The physically accurate simulation of advection dominated processes is a challenging computational problem. Advection is the main driver in the simulation of fluid motion in computational fluid dynamics (CFD), or in the animation of complex geometries using level sets in computer graphics. The difficulty arises from the simultaneous presence of a broad range of length scales (*e.g.* fine geometric features of a 3D model), and their interactions. Presently, the workhorse approach in simulating multiscale flow phenomena such as turbulent flows are Direct Numerical Simulations (DNS) [1] which use large uniform grids to resolve all scales of the flow. Large DNS calculations are performed on massively-parallel computing architectures and compute the evolution of hundreds of billions of unknowns [23].

DNS is not a viable solution for the simulation of flows of engineering interest [4] albeit the continuous increase in available high performance computers. Adaptive simulation techniques, such as Adaptive Mesh Refinement (AMR) [5], or multiresolution techniques using Wavelets [6][7][8] have been introduced in order to locally adjust the resolution of the computational elements to the different

length scales emerging in the flow field. Wavelets have been employed largely for conservation laws but they have also recently been coupled with level sets for geometry representation [9].

In order to create simulation tools that go beyond the state-of-the-art, these adaptive numerical methods must be complemented with massively-parallel and multi-level parallel scalability. Hardware-accelerated wavelet transforms have a long history. The first GPU-based 2D Fast Wavelet Transform was introduced by Hopf and Ertl in 2000 [10]. Since then many different parallel implementations have been proposed and evaluated [11] both on multi-core architectures such as the Cell BE [12], and GPUs [13]. These efforts however, have been restricted to the *full* FWT. The effective parallel implementation of *adaptive* wavelet-based methods is however hindered by their inherently sequential nested structure. This difficulty limits their effective implementation on multi-core and many-core architectures and affects the development of per-thread-based wavelet software that can have both high performance and abstracts from a specific hardware architecture. An alternative approach for the effective implementation of wavelets for flow simulations is to specify parallel tasks instead of threads and then let an external library map logical tasks to physical threads according to the specific hardware architecture. Another emerging need in developing simulation software is the necessity to specify more than one granularity level for parallel tasks in order to combine multi-core computing with many-core accelerators such as GPUs.

In this paper we present multiresolution wavelet based computational methods designed for different multi-core architectures. The resulting computational framework, is flexible and can be used to simulate different natural phenomena such as transport of interfaces, reaction-diffusion, or compressible flows.

The paper is organized as follows: first we introduce briefly wavelets, followed by the presentation of our algorithms for the discretization of partial differential equations. We discuss how the framework can be employed for multi-core and SMP machines. The performance of the proposed simulation tools is then demonstrated on computations of level set based advection of interfaces and two dimensional compressible flows.

2 Wavelet-Based Grids

Biorthogonal wavelets can be used to construct multiresolution analysis (MRA) of the quantities being represented and they are combined with finite difference/volume approximations to discretize the governing equations. Biorthogonal wavelets are a generalization of orthogonal wavelets and they can have associated scaling functions that are symmetric and smooth [14]. Biorthogonal wavelets introduce two pairs of functions, ϕ, ψ for synthesis, and $\tilde{\phi}, \tilde{\psi}$ for analysis.

There are four refinement equations:

$$\phi(x) = \sum_m h_m^S \phi(2x + m), \quad \psi(x) = \sum_m g_m^S \phi(2x + m), \quad (1)$$

$$\tilde{\phi}(x) = \sum_m h_m^A \tilde{\phi}(2x + m), \quad \tilde{\psi}(x) = \sum_m g_m^A \tilde{\phi}(2x + m). \quad (2)$$

Given a function f , we compute $c_k^0 = \langle f, \tilde{\phi}_k^0 \rangle$, $d_k^l = \langle f, \tilde{\psi}_k^l \rangle$ and reconstruct f as:

$$f = \sum_k c_k^0 \phi_k^0 + \sum_{l=0}^{\infty} \sum_k d_k^l \psi_k^l. \quad (3)$$

The Fast Wavelet Transform (FWT) uses the filters h_n^A, g_n^A (analysis), whereas h_n^S, g_n^S are used in the inverse fast wavelet transform (synthesis):

$$c_k^l = \sum_m h_{2k-m}^A c_m^{l+1}, \quad d_k^l = \sum_m g_{2k-m}^A c_m^{l+1}, \quad (4)$$

$$c_k^{l+1} = \sum_m h_{2m-k}^S c_m^l + \sum_m g_{2m-k}^S d_m^l. \quad (5)$$

Active scaling coefficients Using the FWT we can decompose functions into scaling and detail coefficients, resulting in a MRA of our data. We can now exploit the scale information of the MRA to obtain a compressed representation by keeping only the coefficients that carry significant information:

$$f_{\geq \varepsilon} = \sum_k c_k^0 \phi_k^0 + \sum_l \sum_{k: |d_k| > \varepsilon} d_k^l \psi_k^l, \quad (6)$$

where ε is called threshold and is used to truncate terms in the reconstruction. The scaling coefficients c_k^l needed to compute $d_k^l : |d_k^l| > \varepsilon$, and the coefficients at coarser levels needed to reconstruct c_k^l are the active scaling coefficients. The pointwise error introduced by this thresholding is bounded by ε . Each scaling coefficient has a physical position. Therefore the above compression results in an adapted grid \mathcal{K} , where each grid node will represent an active scaling coefficient, representing a physical quantity. We then discretize the differential operators by applying standard finite-volume or finite-difference schemes on the active coefficients. One way to retain simple operations is to first create a local, uniform resolution neighborhood for a grid point, and then apply the operator on it. Such operators can be viewed as (non-linear) filtering operations on uniform resolution grid points, formally:

$$F(\{c_{k'}^l\}_{k' \in \mathbb{Z}})_k = \sum_{j=s_f}^{e_f-1} c_{k+j}^l \beta_j, \quad \beta_j^l \text{ function of } \{c_m^l\} \quad (7)$$

where $\{s_f, e_f-1\}$ is the support of the filter in the index space and $e_f - s_f$ is the number of non-zero filter coefficients $\{\beta_j\}$. In order to perform uniform resolution filtering we need to temporarily introduce artificial auxiliary grid points, so-called “ghosts”. We need to ascertain that for every grid point k in the adapted set \mathcal{K} , its neighborhood $[k - k_f, k + k_f]$ is filled with either other points k' in \mathcal{K} or ghosts g . Using this set of ghosts, which can be precomputed and stored or computed on the fly (see section 2), we are now able to apply the filter F to all points k in \mathcal{K} . The ghosts are constructed from the active scaling coefficients as

a weighted average $g_i^l = \sum_l \sum_j w_{ijl} c_j^l$, where the weights w_{ijl} are provided by the refinement equations (4). It is convenient to represent the construction of a ghost as $g_i = \sum_j w_{ij} p_j$, where i is the identifier for the ghost and j represents the identifier for the source point p_j which is an active scaling coefficient in the grid. Calculation of the weights $\{w_{ij}\}$ is done by traversing a dependency graph associated with the refinement equations (see Figure 1). This operation can be expensive for two reasons: firstly if the dependency graph has loops, we need to solve a linear system of equations to compute $\{w_{ij}\}$, secondly, the complexity of calculating the values $\{w_{ij}\}$ scales with the number of dependency edges *i.e.* it grows exponentially with the jump in level between a point k and its neighbors in the adapted grid. The wavelets incorporated into the present framework are based on subdivision schemes and either interpolate function values or their averages [15]. Due to their construction, these wavelets do not exist explicitly in an analytic form. The primal scaling function can however be constructed by a recursive scheme imposed by its refinement equation. By virtue of their interpolatory property, the ghost reconstruction is straightforward (the ghost dependency graphs do not contain loops) and leads to very efficient reconstruction formulae.

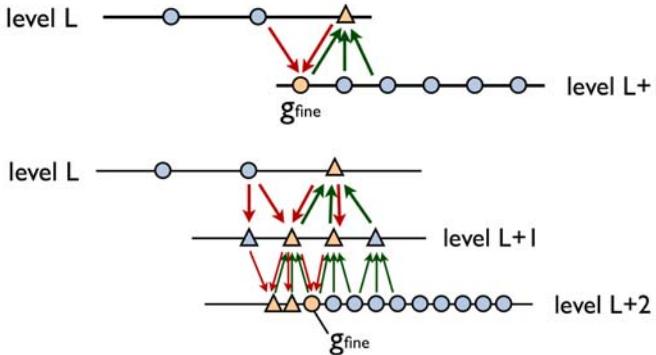


Fig. 1. Dependency graph for the ghost reconstruction g_{fine} with a jump in resolution of 1 (top) and 2 (bottom), for the case of a linear, non-interpolating, biorthogonal wavelet. The ghost generates secondary (triangles), temporary ghosts to calculate the needed weights and source points. The unknown ghosts are denoted in orange.

2.1 Using Wavelet-Blocks

The wavelet adapted grids for finite differences/volumes are often implemented with quad-tree or oct-tree structures whose leaves are single scaling coefficients. The main advantage of such fine-grained trees is the very high compression rate which can be achieved when thresholding individual detail coefficients. The drawback on the other hand is the large amount of sequential operations they involve and the number of indirections (or read instructions) they need in order to access a group of elements. Even in cases where we only compute without

changing the structure of the grid, these grids already perform a great number of neighborhood look-ups. In addition, operations like refining or coarsening single grid points have to be performed and those operations are relatively complex and strictly sequential. To expose more parallelism and to decrease the amount of sequential operations per grid point, the key idea is to simplify the data-structure. We address this issue by decreasing the granularity of the method at the expense of a reduction in the compression rate. We introduce the concept of block of grid points, which has a coarser granularity by one or two order of magnitude with respect to the number of scaling coefficients in every direction *i.e.* in 3D, the granularity of the block is coarser by 3 to 5 orders of magnitude with respect to a single scaling coefficient.

Structure of the Wavelet-Blocks. A block contains scaling coefficients which reside on the same level and every block contains the same number of scaling coefficients. The grid is then represented with a tree which contains blocks as leaves (see Figure 2). The blocks are nested so that every block can be split and doubled in each direction and blocks can be collapsed into one. Blocks are interconnected through the ghosts. In the physical space the blocks have varying size and therefore different resolutions. The idea of introducing the intermediate concept of a block provides a series of benefits. The first benefit is that tree operations are now drastically accelerated as they can be performed in $\log_2(N^{1/D}/s_{block})$ operations instead of $\log_2(N^{1/D})$, where N is the total number of active coefficient, D the dimensions in consideration and s_{block} is the block size. The second benefit is that the random access at elements inside a block can be extremely efficient because the block represents the atomic element. Another very important advantage is the reduction of the sequential operations involved in processing a local group of scaling coefficients. We consider the cost c (in terms of memory access) of filtering per grid point with a filter of size KD ($c = KD$). In a uniform resolution grid the data access operations to perform the computation is proportional to KD . For a fine-grid tree the number of accesses is proportional to $c = KD \log_2(N^{1/D})$. Using the wavelet-blocks approach and assuming that s_{block} is roughly one order of magnitude larger than K , the ratio of ghosts needed to perform the filtering for a block is

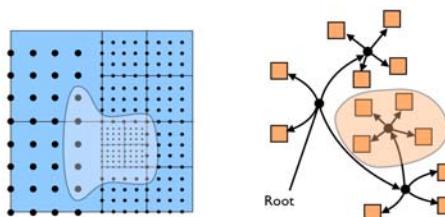


Fig. 2. The masked (left) region identifies the source points of the grid used to construct the ghosts, which are used to collapse/split the blocks in the tree (right masked region)

$$\begin{aligned}
r &= \frac{(s_{block} + K)^D - s_{block}^D}{s_{block}^D} \approx D \frac{K}{s_{block}} \\
c(K) &= (1 - r)KD + rKD(\log_2(N^{1/D}/s_{block})) \\
&= KD + KDr(\log_2(N^{1/D}/s_{block}) - 1).
\end{aligned} \tag{8}$$

We assume that none of the blocks is overlapping in the physical space. We improve the efficiency of finding the neighbors of a block by constraining the neighbors to be adjacent neighbors. Because of this constraint, the additional condition $s_{block} \geq w_{stencil}2^{L_j}$ appears, where $w_{stencil}$ is the filter size and L_j is the maximum jump in resolution.

Local compression and refinement of the grid. The re-adaptation of the grid is achieved by performing elementary operations on the blocks: block splitting to locally refine the grid, and block collapsing to coarsen the grid. Block splitting and block collapsing are triggered by logic expressions based on the thresholding of detail coefficients residing inside the block. To compute the detail coefficients of the block (i_b, l_b) we perform one step of the fast wavelet transform :

$$d_k^{(l_b-1)} = \sum_m g_{2k-m}^A c_m^{l_b}, \quad k \in \left\{ \frac{i_b \cdot s_{block}}{2}, \frac{(i_b + 1) \cdot s_{block}}{2} - 1 \right\} \tag{9}$$

Note that k is inside the block but m can be outside the block (see Figure 2). In the one dimensional case, when we decide to collapse some blocks into one, we just have to replace the data with the scaling coefficient at the coarser level:

$$c_k^l = \sum_m h_{2k-m}^A c_m^{l+1} \quad k \in \left\{ \frac{i_b}{2} \cdot s_{block}, \frac{(i_b + 1)}{2} \cdot s_{block} - 1 \right\} \tag{10}$$

The complexity of a single FWT step, which consists of computing all the details and the scaling coefficients, is approximately $D \cdot (c \cdot s_{block}/2)^D$ with $c = c(K)$ defined as in Equation 8 and K is the maximum stencil size of the FWT filters. If we consider the block collapse as the key operation for compressing, we can consider the block splitting as the key to capture smaller emerging scales. In the case where we split one block into two blocks, we perform one step of the inverse wavelet transform on the block (i_b, l_b) :

$$c_k^{l_b+1} = \sum_m h_{2m-k}^S c_m^{l_b}, \quad k \in \{2i_b \cdot s_{block}, 2(i_b + 1) \cdot s_{block} - 1\} \tag{11}$$

Time refinement. Small spatial scales are often to be associated with small scales in time, especially if the equation under investigation contains nonlinear terms. The wavelets-adapted block grid can exploit time refinement [16][17]. Depending on the case, this can already accelerate the simulation by one or two orders of magnitude. There are however two shortcomings in exploiting different time scales. The first drawback comes from the fact that the processing of the grid blocks has to be grouped by their time scales and no more than one group can be processed in parallel. The second drawback is the increased complexity of reconstructing ghosts; when dealing with different time scales, ghosts have to be interpolated also in time.

3 Wavelet Algorithms for Multi-core Computing

Our first objective in designing the framework is to expose enough load balanced parallel tasks so that the amount of tasks is greater than the number of cores. The second objective is to express nested parallelism inside the parallel tasks, so that the idling cores could help in processing the latest tasks. Once these are met, we expect an external library to map our logical tasks into physical threads based on the specific SMP architecture. Grid adaptation is performed in two steps: before computing we refine the grid in order to allow the emergence of new smaller scales [18]. After the computation we apply compression based on thresholding to retain only blocks with significant detail. Figure 3.2 shows the program flow for the refinement stage. An important aspect of the framework is its ability to retain control over the maximum number of scaling coefficients and therefore, the memory; instead of keeping the threshold fixed, we can also control the number of scaling coefficients with an upper bound by changing the compression threshold.

3.1 Producing Many Parallel Tasks

Due to the design of the framework, the number of parallel tasks scales with the number of blocks. Parallel tasks can be grouped into *elementary* and *complex* tasks. A task of the *elementary* type operates exclusively on data inside the block, *i.e.* it is purely local. These tasks are inherently load-balanced, since every task operates on the same number of points. A *complex* task, however, involves the reconstruction of ghosts and block splitting, the FWT, or evaluating finite-difference operators. The cost of reconstructing a ghost is not constant and can be expensive as it depends on the structure of the tree. The efficiency of the ghost reconstruction is therefore paramount for load balance.

3.2 Fast Ghost Reconstruction

We recall that ghosts are computed as weighted averages of the scaling coefficients in its neighborhood. We call n_i the number of source points/weights that we need to construct the ghost g_i from:

$$g_i = \sum_j w_{ij} p_j, \quad n_i = |\{j \in \mathbb{Z} : w_{ij} \neq 0\}|. \quad (12)$$

We can split the ghost construction into two stages: find the contributing source points/weights and evaluate the weighted average. The first stage will produce an array of weights and indices associated with grid points, which is successively sent as input to the second stage where the array will be used to produce the weighted summation. We call this array the instruction array. We note immediately that the first stage is computationally expensive because it needs recursive data structures. If we assume that we have to construct the same ghost several times, it is meaningful to store the instructions array and successively perform

only the second stage, which is fast. For the same ghost of a block, we can use the same instructions array as long as the neighbor blocks do not change. The main shortcoming of this strategy is its memory consumption, as the instructions arrays tend to be big. However, the entries of the instruction arrays of two close ghosts are likely to be very similar. This coherence between instructions arrays allows us to compress them before they are saved. To compress them, we re-organize every instruction array of a block into streams of w_{ij} 's, j 's and n_i 's, and then pass them to an encoder (using either quantization based encoding or Huffman encoding). We noticed that overall, the compression factor of the instruction array varies between 1.5 and 5, and the compression of n_i is between 10 and 20.

4 Results

In this section we present the tools used to develop the framework and the performance of our algorithms on a set of benchmark problems. We then present a study of the effect of different block sizes, types of wavelets and the number of threads.

Software Details. The framework was written in C++ using generic programming and object oriented concepts. In our framework, Intel's Threading Building Blocks (TBB) [19] library was used to map logical tasks to physical threads. This library completely fulfills our requirements stated in Section 3, as it allows to specify task patterns and enables to easily express nested parallelism inside tasks. Furthermore TBB provides a very useful set of templates for programming in parallel which are independent of the simulated phenomena. Most of the TBB calls were to `parallel_for`, which exploits recursive parallelism. We employed the `auto_partitioner` for an automatic grain size through the 'reflection on work stealing' [20], to dynamically change the grain size of tasks. In fact, the most efficient grain size varies with respect to the block size and the wavelet order but also with the number of cores.

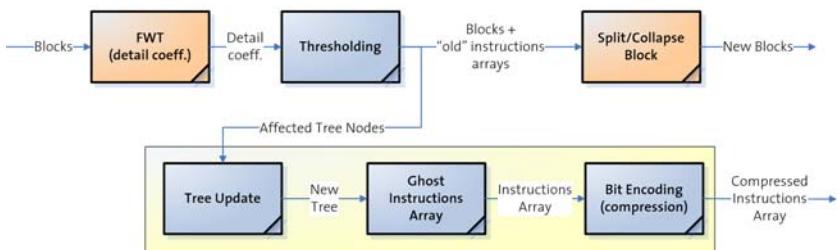


Fig. 3. Stages of refinement/compression of the grid. The region depicted in yellow is performed sequentially but executed in parallel for different blocks. The orange stages need to be optimally parallelized as they involve intensive processing of the data.

Computer architectures. Simulations were done on the ETH Zurich central cluster “Brutus”, a heterogeneous system with a total of 2200 processors in 756 compute nodes. The compute nodes that meet our needs *i.e.* multi-threaded nodes are the eight so-called *fat* nodes each with eight dual-core AMD Opteron 8220 CPU’s (2.8 GHz) and 64-128 GB of RAM. These nodes are inter-connected via a Gigabit Ethernet backbone. The Intel C++ compiler v10.1.015 (64-bit) was used along with the Intel TBB library (64-bit) to build the three different test cases used in this work. For some of the cases, we have also used an Intel Xeon 5150 machine, with 2 dual core processors with a core speed of 2.66 GHz and 4GB RAM.

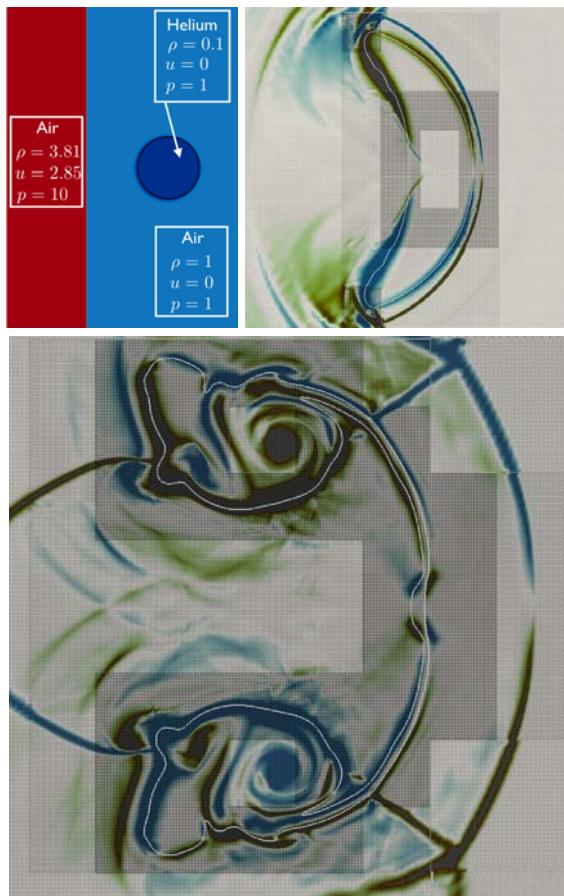


Fig. 4. Initial condition (top left) and zooms of the simulation to the high resolution region close to the bubble at different times (chronologically: top right, bottom). Blue/green depicts high positive/negative vorticity, whereas the interface of the bubble is depicted in white.

4.1 Benchmark Problems

Two-dimensional Compressible Flows. A common benchmark problem for compressible flow simulations consists of a circular bubble filled with helium, surrounded by air being hit by a shock wave (see Figure 4 top left). The shock deforms the bubble and leads to instabilities and fine structures on the helium-air interface. The governing equations are:

$$\begin{aligned} \frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) &= 0 \\ \frac{\partial \rho \mathbf{u}}{\partial t} + \nabla \cdot (\rho \mathbf{u} \otimes \mathbf{u} + p \mathbf{I}) &= 0 \\ \frac{\partial \rho E}{\partial t} + \nabla \cdot ((\rho E + p) \mathbf{u}) &= 0 \end{aligned} \quad (13)$$

with ρ , μ , \mathbf{u} , p and E being the fluid density, viscosity, velocity, pressure and total energy respectively. This system of equations is closed with an equation of state of the form $p = p(\rho)$. In this test case, each grid point stored the values ρ , ρu , ρv , ρE . The simulation was performed using a maximum effective resolution of 4096×4096 grid points with third order average interpolating wavelets, a block size of 32, and a maximum jump in resolution of 2. Derivatives and numerical fluxes were calculated using 3rd order WENO [21] and HLLE [22] schemes respectively. Time stepping was achieved through a 3rd order low storage TVD Runge-kutta scheme. The thresholding was a unified thresholding based on the bubble level set and the magnitude of the velocity. The simulation was computed on the Xeon machine with 4 cores, with a strong speedup of 3.74, that decreases to 3.55 when there are less than 80 blocks. The compression rate varied between 10 and 50.

Three-dimensional Advection of Level sets. We simulated the advection of level sets in 3D based on the advection equation $\frac{\partial \phi}{\partial t} + \mathbf{u} \cdot \nabla \phi = 0$. Derivatives

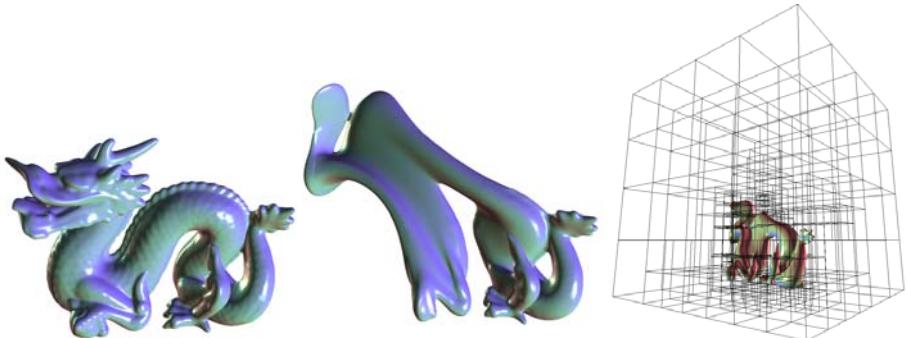


Fig. 5. Initial interface (left), advected with a velocity field of a vortex ring (middle) and distribution of the blocks in the grid (right)

and numerical fluxes were constructed using a 5th order WENO [21] and a Godunov flux respectively. Time integration was performed with a first-order time refinement, where given a fixed CFL number, each block is updated based on its own time step, *i.e.* for a block at level l we used a time step proportional to 2^{-l} . CFL number is defined as:

$$\text{CFL} = \Delta t \frac{|\mathbf{u}|_{max}}{\Delta x_{\text{local}}} \quad (14)$$

with $\Delta x_{\text{local}} = 2^{-l} \frac{1}{\text{block size}}$.

The level set field was also re-distanced after each time step using the re-initialization equation $\frac{\partial \phi}{\partial t} + \text{sgn}(\phi_0)(|\nabla \phi| - 1) = 0$, proposed by Sussman, Smereka and Osher [23] with ϕ_0 being the initial condition of the re-initialization equation

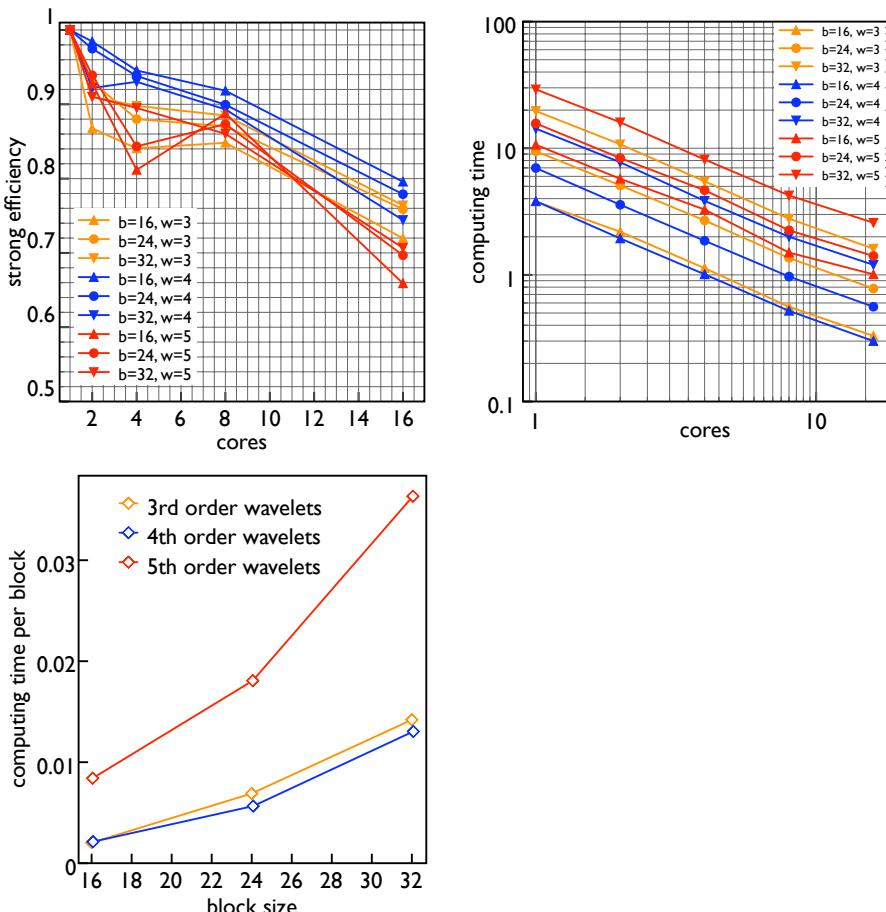


Fig. 6. Efficiency (left), computing times (middle) and time spent per block (right) for different interpolating/average interpolating wavelets and block sizes

which in general, does not have the desired signed distance property. For efficient level set compression we employed the detail-clipping trick introduced in [9]. As initial interface we used the dragon illustrated in Figure 5. Simulations were carried out for four different types of wavelets: 3rd and 5th order average interpolating wavelets, 2nd and 4th order interpolating wavelets, and for different block sizes ranging from 16 to 32 as well as for a range of cores from 1 to 16, while $\varepsilon_{\text{compression}}$ and $\varepsilon_{\text{refine}}$ were kept fixed. We measured the time spent in the computing stage. Figure 6 shows the strong efficiency obtained from these measurements. The largest decrease in performance is from 8 to 16 cores. In the case of 16 cores, the best result was achieved using a block size of 16 with the interpolating wavelet of order 4. With this settings we achieved a strong efficiency of 0.8, with a strong speedup of 12.75 with 16 cores. While with the fourth order interpolating wavelets we achieved a monotonic decreasing efficiency, the efficiency obtained with average interpolating wavelets of order 3 and 5 was better with 8 cores than with 4. We observe that the combination of fourth order interpolating wavelets with a block size of 16 also achieved the best timings (see Figure 6). Furthermore, we note that the lines do not cross: the change in the number of cores does not affect the ranking of the computing time for the different wavelets and block size. We also note that the second fastest setting was the third order average interpolating wavelets with a blocksize of 16, which did not show a good efficiency (0.72 with 16 cores). Regarding the time spent per block versus the block size, the best timing was achieved by the third order average interpolating wavelets. Theoretically, the time spent per block should increase roughly by a factor of 8 between a block size of 16 and 32. We measure an increase by a factor of about 7 for all three wavelets, meaning that for a block size of 16, the “pure computing” time per block was about 85% of the total time spent per block. We must however note that the time spent per block also includes some synchronization mechanism, which scales with the number of blocks which is high in the case of the third order average interpolating wavelets with a blocksize of 32 (Figure 6). Figure 7 depicts the time spent versus the

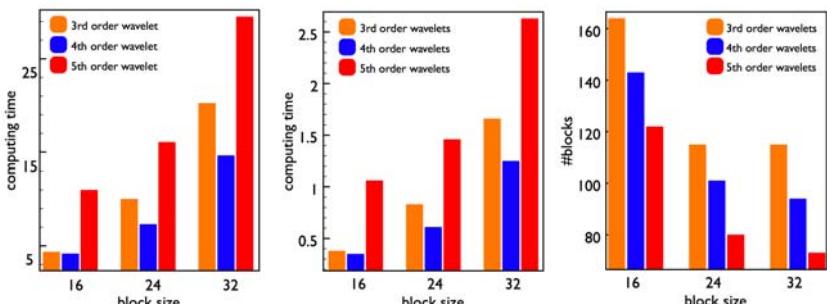


Fig. 7. Computing time versus different block sizes by using one core (left), 16 cores (middle) and the number of blocks (right)

block size with one core (left) and 16 cores (middle). We can see that with the third order wavelet we have a substantial decrease in performances with a block size of 32; this is due to the high number of blocks (Figure 6, right) and the cost associated with the reconstruction of the ghosts at the coarser resolutions.

5 Conclusions

We presented a wavelet-based framework for adaptive multiresolution simulations for solving time-dependent PDEs. The present framework demonstrates that it is possible to develop multiresolution simulation tools that can exploit the new multi-core technologies without degrading the parallel scaling. We have also shown the change in performance and compression rate based on the choice of the wavelets, the size of the block and the number of cores. We achieved the best performances in terms of scaling and timings using a block size of 16, the best wavelet was the fourth-order interpolating one, which gave a strong speedup of 12.75 of 16 cores. We observed that bigger block sizes can be efficiently coupled with high order wavelets, whereas smaller block sizes are more efficient when combined with small block sizes. We have also seen that the performance differences between combinations of block sizes and wavelets are independent of the number of cores, meaning that good efficiency is expected also for more than 16 cores. The current work includes an extension of the framework to support particle-based methods [24] and the incorporation of multiple GPUs as computing accelerators. On the application level we focus on extending the two-dimensional flow simulations to three-dimensional flows and on combining flow simulations and reaction-diffusion processes for adaptive simulations of combustion.

References

1. Ishihara, T., Gotoh, T., Kaneda, Y.: Study of High-Reynolds Number Isotropic Turbulence by Direct Numerical Simulation. *Annual Review of Fluid Mechanics* 41(1) (2009)
2. Yokokawa, M., Uno, A., Ishihara, T., Kaneda, Y.: 16.4-Tflops DNS of turbulence by Fourier spectral method on the Earth Simulator. In: *Supercomputing 2002: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pp. 1–17. IEEE Computer Society Press, Los Alamitos (2002)
3. Chatelain, P., Curioni, A., Bergdorf, M., Rossinelli, D., Andreoni, W., Koumoutsakos, P.: Billion Vortex Particle direct numerical simulation of aircraft wakes. *Comput. Methods Appl. Mech. Engrg.* 197(13–16), 1296–1304 (2008)
4. Jiménez, J.: Computing high-Reynolds-number turbulence: will simulations ever replace experiments? *Journal of Turbulence* 4 (June 12, 2003); 5th International Symposium on Engineering Turbulence Modelling and Measurements, Mallorca, Spain, September 16–18 (2002)
5. Berger, M.J., Oliger, J.: Adaptive mesh refinement for hyperbolic partial differential equations. *J. Comput. Phys.* 53(3), 484–512 (1984)
6. Harten, A.: Adaptive Multiresolution Schemes For Shock Computations. *Journal of Computational Physics* 115(2), 319–338 (1994)

7. Kevlahan, N.K.R., Vasilyev, O.V.: An adaptive wavelet collocation method for fluid-structure interaction at high reynolds numbers. *SIAM Journal on Scientific Computing* 26(6), 1894–1915 (2005)
8. Liu, Q., Vasilyev, O.: A Brinkman penalization method for compressible flows in complex geometries. *Journal of Computational Physics* 227(2), 946–966 (2007)
9. Bergdorf, M., Koumoutsakos, P.: A Lagrangian particle-wavelet method. *Multiscale Modeling and Simulation* 5(3), 980–995 (2006)
10. Hopf, M., Ertl, T.: Hardware accelerated wavelet transformations. In: *Proceedings of EG/IEEE TCVG Symposium on Visualization VisSym 2000*, pp. 93–103 (2000)
11. Yang, L.H., Misra, M.: Coarse-grained parallel algorithms for multi-dimensional wavelet transforms. *Journal of Supercomputing* 12(1-2), 99–118 (1998)
12. Bader, D.A., Agarwal, V., Kang, S.: Computing discrete transforms on the cell broadband engine. *Parallel Computing* 35(3), 119–137 (2009)
13. Tenllado, C., Setoain, J., Prieto, M., Pinuel, L., Tirado, F.: Parallel implementation of the 2d discrete wavelet transform on graphics processing units: Filter bank versus lifting. *IEEE Transaction on Parallel and Distributed Systems* 19(3), 299–310 (2008)
14. Cohen, A., Daubechies, I., Feauveau, J.: Biorthogonal Bases of Compactly Supported Wavelets. *Communication on Pure and Applied Mathematics* 45(5), 485–560 (1992)
15. Donoho, D.: Interpolating wavelet transforms. Technical report, Preprint Department of Statistics, Stanford University (1992)
16. Ferm, L., Lötstedt, P.: Space–Time Adaptive Solution of First Order PDEs. *J. Sci. Comput.* 26(1), 83–110 (2006)
17. Jansson, J., Logg, A.: Algorithms and data structures for multi-adaptive time-stepping. *ACM Transactions on Mathematical Software* 35(3), 17 (2008)
18. Liandrat, J., Tchamitchian, P.: Resolution of the 1D regularized Burgers equation using a spatial wavelet approximation. Technical Report 90-83, 1CASE, NASA Contractor Report 18748880 (December 1990)
19. Contreras, G., Martonosi, M.: Characterizing and improving the performance of Intel threading building blocks. In: *2008 IEEE International Symposium on Workload Characterization (IISWC)*, Piscataway, NJ, USA, pp. 57–66. IEEE, Los Alamitos (2008)
20. Robison, A., Voss, M., Kukanov, A.: Optimization via reflection on work stealing in TBB. In: *2008 IEEE International Symposium on Parallel & Distributed Processing*, New York, NY, USA, vols. 1-8, pp. 598–605. IEEE, Los Alamitos (2008)
21. Harten, A., Engquist, B., Osher, S., Chakravarthy, S.: Uniformly high order accurate essentially non-oscillatory schemes.3 (Reprinted from *Journal of Computational Physics*, vol 71, p. 231 (1987). *Journal of Computational Physics* 131(1), 3–47 (February 1997)
22. Wendroff, B.: Approximate Riemann solvers, Godunov schemes and contact discontinuities. In: Toro, E.F. (ed.) *Godunov Methods: Theory and Applications*, 233 Spring St, New York, NY 10013 USA, London Math Soc., pp. 1023–1056. Kluwer Academic/Plenum Publ. (2001)
23. Sussman, M., Smereka, P., Osher, S.: A Level Set Approach for Computing Solutions To Incompressible 2-Phase Flow. *Journal of Computational Physics* 114(1), 146–159 (1994)
24. Koumoutsakos, P.: Multiscale flow simulations using particles. *Annual Review Of Fluid Mechanics* 37, 457–487 (2005)

Localized Parallel Algorithm for Bubble Coalescence in Free Surface Lattice-Boltzmann Method

Stefan Donath, Christian Feichtinger, Thomas Pohl, Jan Götz, and Ulrich Rüde

Friedrich-Alexander University Erlangen-Nuremberg,
Chair for System Simulation (LSS), 91058 Erlangen, Germany
`stefan.donath@informatik.uni-erlangen.de`
<http://www10.informatik.uni-erlangen.de/~stefan/>

Abstract. The lattice Boltzmann method is a popular method from computational fluid dynamics. An extension of this method handling liquid flows with free surfaces can be used to simulate bubbly flows. It is based on a volume-of-fluids approach and an explicit tracking of the interface, including a reconstruction of the curvature to model surface tension. When this algorithm is parallelized, complicated data exchange is required, in particular when bubbles extend across several subdomains and when topological changes occur through coalescence of bubbles. In a previous implementation this was handled by using all-to-all MPI communication in each time step, restricting the scalability of the simulations to a moderate parallelism on a small number of processors. In this paper, a new parallel bubble merge algorithm will be introduced that communicates updates of the bubble status only locally in a restricted neighborhood. This results in better scalability and is suitable for massive parallelism. The algorithm has been implemented in the lattice Boltzmann software framework *wALBerla*, resulting in parallel efficiency of 90% on up to 4080 cores.

1 Introduction

Free surface flow is omnipresent in nature, in everyday life as well as in technological processes. Examples are river flow, rain, and filling of cavities with liquids or foams.

A variety of computational fluid dynamics (CFD) techniques have been developed to study multiphase flow phenomena, such as the volume-of-fluid method [1,2,3,4], level-set approaches [5,6,7], boundary integral methods [8,9], and the lattice Boltzmann method (LBM). The advantage of LBM becomes apparent for flows in complex geometries, since the mapping to the lattice is very flexible and involves little computational time. For multiphase flows, various LBM-based numerical models have been developed. Gunstensen et al. [10] proposed a color model for simulation of immiscible fluids based on the model of Rothmann and Keller [11]. Tölke developed a Rothmann-Keller-based model for high density ratios [12]. Shan and Chen presented a LBM with mean-field interactions for



Fig. 1. Simulation of metal foaming with free surface LBM

multiphase and multicomponent fluid flows [13]. Inamuro et al. [14] proposed a LBM for multicomponent immiscible fluid with the same density. There are also methods based on free energy approach [15,16] and level sets [17]. Besides the multiphase and multicomponent models, a couple of free surface models arose in the past decade for simulating moving interfaces between immiscible gas and liquids. The method proposed by Ginzburg and Steiner [18] computes collision only in “active” cells and uses a recoloring operator for redistribution of fluid mass. The variant by Körner et al. [19] is based on the assumption that the influence of gas phase on liquid phase can be reduced to the force exerted by its pressure and the surface tension. A simplified approach was used by Thürey et al. [20,21,22] who used a normalized atmospheric pressure for reconstruction at the interface. Xing et al. [23] reduced the method such that curvature calculation and surface reconstruction can be omitted. The advantage of Körner’s method is that simulation of bubbly flows and even foams is possible, and thus Pohl [24] implemented it for three dimensions and successfully altered the algorithm such that efficiency of MPI parallelization was improved. Further performance optimization efforts [25,26] made simulations of real engineering applications like metal foaming processes feasible (see Fig. 1). Our group looks back on seven years of experience in free surface LBM [27,22]. This study further increases the usability of the parallel free surface solver by proposing a new communication strategy making it adequate for thousands of processors. The novel algorithm is described and compared to those presented in [24].

2 Free Surface Lattice Boltzmann Method

The LBM [28] is a mesoscopic CFD simulation scheme that is based on kinetic fluid theory and uses a stencil-based approach on a Cartesian grid, to solve time-dependent quasi-incompressible flows in continuum mechanics. It operates on *particle distribution functions* (PDFs) that represent the fraction of the total mass in each lattice that moves in a discrete direction. A popular model for the collision process is the BGK approximation where the evolution equation reads with $i = 0, \dots, N_e - 1$:

$$f_i(\mathbf{x} + \mathbf{e}_i \Delta t, t + \Delta t) = f_i(\mathbf{x}, t) - \frac{\Delta t}{\tau} \left[f_i(\mathbf{x}, t) - f_i^{eq}(\rho(\mathbf{x}, t), \mathbf{u}(\mathbf{x}, t)) \right]. \quad (1)$$

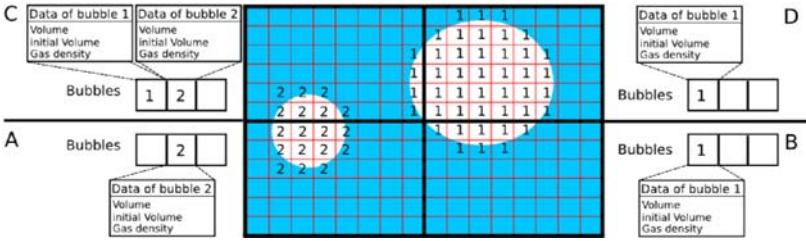


Fig. 2. Data representation of a bubble in free-surface LBM: Every gas and interface cell holds an ID of the bubble it belongs to. Besides this cell-based data, each process stores a list of bubble entries containing the corresponding volume data for each bubble.

Here, f_i is the PDF, representing the fraction of particles located at position \mathbf{x} and time t , moving into the discrete velocity direction \mathbf{e}_i . Time step Δt and length of the discrete velocity vectors \mathbf{e}_i are chosen such that the particle distributions move exactly one cell further during each time step. The difference from the equilibrium f_i^{eq} is weighted with the relaxation time τ which depends on the viscosity of the fluid. The equilibrium is a function of the macroscopic density ($\rho = \sum_0^{N_e-1} f_i$) and velocity ($\rho \mathbf{u} = \sum_0^{N_e-1} f_i \mathbf{e}_i$).

The free surface extension introduces different cell types for gas, liquid and the interface in between. For liquid cells, standard LBM is performed, while in gas cells no collision takes place. For all cells, a fill value is introduced which specifies the normalized quantity of liquid in a cell. Gas and interface cells store an identifier (ID) of the region of gas (*bubble*) they belong to (see Fig. 2). Conversion rules ensure closure of interface during movement, prohibiting direct adjacency of gas and liquid cells. In interface cells, PDFs are missing from the gas phase and thus have to be reconstructed such that the velocities of both phases are equal, and the force performed by the gas is balanced by the fluid's force. To realize this, all PDFs with $\mathbf{e}_i \cdot \mathbf{n} < 0$ are computed like

$$f_i(\mathbf{x} - \mathbf{e}_i \Delta t, t + \Delta t) = f_i^{eq}(\rho_G, \mathbf{u}) + f_{\bar{i}}^{eq}(\rho_G, \mathbf{u}) - f_{\bar{i}}(\mathbf{x}, t), \quad (2)$$

where $f_{\bar{i}}$ points into the opposite direction of f_i , i.e. $\mathbf{e}_{\bar{i}} = -\mathbf{e}_i$. The gas density is calculated by $\rho_G = 3p_G + 6\sigma\kappa(\mathbf{x}, t)$, which arises from the energy balance $p_G \cdot dV = \sigma \cdot dA$ with surface tension σ on the surface dA and gas pressure p_G in volume dV . Gas pressure is calculated from the volume of the bubble, implying that any volume change resulting from change of fill values or conversion of cells has to be tracked throughout the simulation. The surface normal \mathbf{n} is computed as the gradient of the fill values using a Parker-Youngs approximation [29]. It is also used for computing surface points for each cell, needed for determination of curvature κ . This is the most computing-intensive part of the method.

For details on the LBM in general see [28]. Details on the free surface extension can be found in [19] and [24].

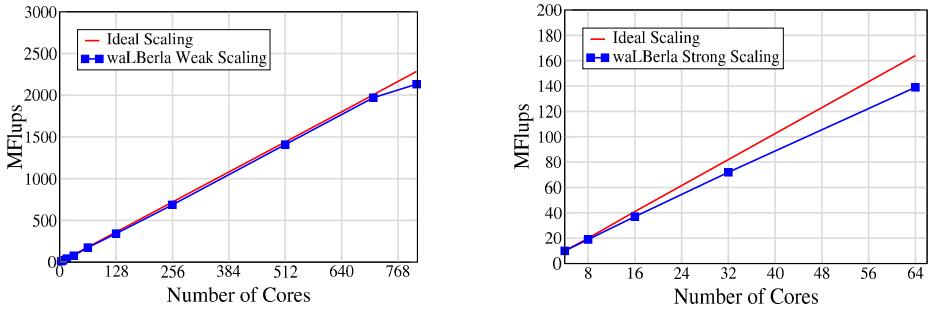


Fig. 3. Weak (left) and strong (right) scaling of waLBerla executing standard LBM single phase simulation on a Woodcrest platform. Performance measured in *million fluid lattice site updates* (see Sec. 5).

3 Software Framework waLBerla

The method has been incorporated into an existing LBM framework called *waLBerla* [30] which features basic tools for data management and parallelization. It splits the computational domain into so-called *patches* that may have different features implemented by the user. Since one process can be assigned to several patches, waLBerla internally realizes data exchange by either process-local or MPI communication. Each data exchange consists strictly of a pair of send and blocking receive, in exactly this order. For each iteration of the time loop, the sequence control executes a user-defined order of so-called *sweep* routines that traverse the grid and manipulate data. Before each sweep, communication of required data takes place. The sweep function can also request a reiteration before the next sweep is executed. The core of waLBerla features basic functionalities for LBM simulation and guarantees good scaling efficiency for 512 cores and beyond (see Fig. 3). More details on waLBerla’s performance can be found in [31]. In the following, “process” and “patch” will be used synonymously, since waLBerla makes the difference between local and MPI communication transparent to the user.

4 Merge Algorithm

Free surface LBM has a high communication-to-computation ratio. During one time step cell-based data like normals, PDFs, and cell states have to be communicated seven times, and bubble volume data once. Cell-based data is communicated only with direct neighbors, but volume data of bubbles has to be broadcasted to a larger vicinity. In Pohl’s algorithm, this data is exchanged among all processes which facilitates handling of bubble coalescence. This approach scales well for a small number of processes. However, if more than 100 processes are involved, all-to-all communication is inefficient. The localized algorithm described in this Section handles merges of bubble volume data such that only processes harboring the same bubble have to communicate the data.

Bubble coalescence occurs if two bubbles are very near to each other and there are no hindering forces. Algorithmically, two gas regions will be merged as soon as the interfaces have contact, i.e. two interface cells of different bubbles are adjacent. Then, the volume data of one bubble is added to the other and all cells of the old bubble are assigned the ID of the target bubble. Of course, this incident has to be communicated throughout the whole domain, or at least throughout the region the two bubbles occupy. In case several bubbles touch each other in the same time step, all bubbles are transformed to a single target bubble (see Fig. 4).

Since in Pohl's method all bubbles' volume data is known to every process, it is possible to efficiently combine data exchange of bubble coalescence with the routinely performed exchange of volume update. Thus, even if several bubble merges occur in a time step, only one all-to-all communication is necessary.

4.1 Arising Challenges Due to Localization

Storing bubbles only on processes they are contained in, is the key to large-scale parallelization. However, this results in several problems. If a bubble crosses the process border, the target process has to recognize that it needs to receive the volume data of a new bubble, and it has to be clear which process sends this data. Furthermore, in case of a merge it may happen that a known bubble merges with other bubbles that are not known to the process (e.g. patch D sees bubble 2 merging with unknown ID 1 in Fig. 4). In the waLBerla implementation, the ID used to connect interface and gas cells to the corresponding bubbles is indicator for the bubble data transmission. When this cell-based data is communicated between neighbors, a process can recognize unknown IDs in the halo layers and await bubble volume data from the neighbor in the next sweep (see Fig. 2). Likewise, if a process recognizes IDs at the border which are not present in the halo, and the corresponding bubble is not yet registered on the neighbor process, it will send this bubble in the next sweep.

In the novel implementation, exchange of volume updates occurs only among those processes a bubble resides on. For this, each bubble holds a list of its harboring patches. Merging of bubbles is split in two steps, the recognition and the merge. If a process recognizes during a sweep that two bubbles have to be merged, this information is stored in the bubble and also exchanged among the owning processes when update data is communicated. This way, conflicts can be solved if a bubble touches another twice on different processes (like bubble 12 and 4 on patches H and J in Fig. 4). Having distributed this information consistently, the algorithm presented in the following subsection is able to process all outstanding merges.

4.2 Merge Sequence

Due to the communication scheme of waLBerla, message exchange occurs before the merge algorithm can start to act. In case of the merge sweep special communication buffers are used which are empty in the beginning. Thus, if no merges

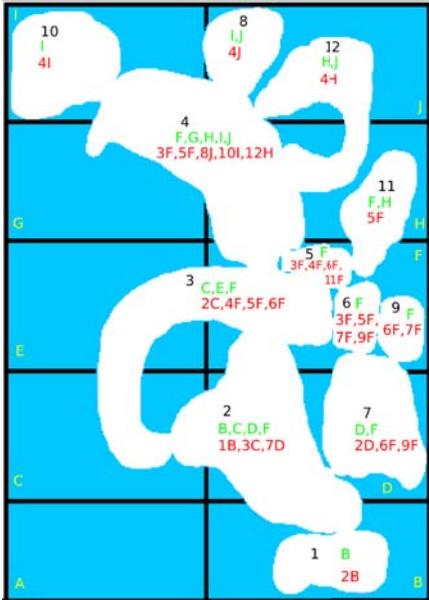


Fig. 4. Bubbles with their IDs (black numbers), lists of patches (green letters) and merge information (red IDs and patches)

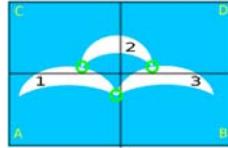


Fig. 5. Sketch of a simple merge example causing an intermediate inconsistent state

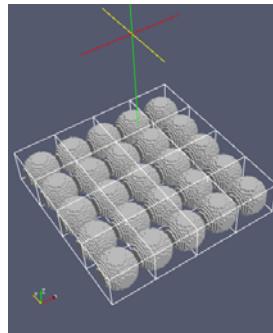


Fig. 6. Worst case example for merge algorithm, where 25 bubbles merge in same time step on 16 processes

are scheduled, no communication overhead will occur (see line 2 in Alg. 1). Based on merge information available, new iterations of this sweep are requested. Consequently, processes with no bubbles involved in merges may already perform the next sweep.

Basically, the algorithm shown in Alg. 1 relies on three principles: For each merge pair exactly one patch is responsible to perform the merge. Merges are performed hierarchically, i.e. action starts with highest bubble IDs rippling down to the lowest, using as many iterations as needed. Patches that recognize outstanding merges, but are not responsible, expect to get appropriate data from the patch in charge.

Lines 4 to 15 ensure that appropriate communication pairs are established such that no deadlock can occur. As soon as a responsible patch does not have to wait for data of higher merges (line 19), it can start to merge the pair of bubbles. Thereby it has to ensure that all other patches knowing one of the involved bubbles are supplied with the required data: It sends an *A-type message* containing complete data of the new bubble to all patches that knew the removed bubble. All patches that know the target bubble get a *B-type message* containing the additional volume. This distinction is important in the case that one of these patches knows both bubbles (like patch F for bubbles 2 and 3, when patch C merges them in Fig. 4). Depending on the order bubbles are merged, the situation may arise that a merge target of a third bubble is not existent anymore. In this

Algorithm 1. The Merge Algorithm

```

1 repeat
2   if communicationBuffers not empty then communicate messages;
3   newIterationRequired  $\leftarrow$  false;
4   foreach bubble B do
5     if B merges on other patch P then
6       communicationBuffers  $\leftarrow$  expect message from P;
7       newIterationRequired  $\leftarrow$  true;
8     end
9     if B merges on this patch
10    and resides also on other patches
11    and B is not in inconsistent state then
12      foreach patch P B resides on do
13        communicationBuffers  $\leftarrow$  empty message to P;
14      end
15      newIterationRequired  $\leftarrow$  true;
16    end
17  end
18  foreach bubble B with scheduled merges do
19    if B merges only with lower IDs
20    and B merges with next lower ID M on this patch
21    and M's highest ID to merge is B then
22      Add volume of B to M;
23      scheduledIDRenamings  $\leftarrow$  B into M;
24      foreach bubble O that is scheduled to merge with B do
25        replace merge to B with merge to M;
26        send C message to all patches O resides on;
27      end
28      add all merge information of B to M;
29      oldMpatchList  $\leftarrow$  current list of patches M resides on;
30      add list of patches B resides on to M;
31      foreach patch P in oldMpatchList do
32        send B message containing update data for M;
33      end
34      oldMpatchList  $\leftarrow$  current list of patches M resides on;
35      foreach patch P in oldMpatchList do
36        send A message containing whole data of M to replace B;
37      end
38      remove bubble data B;
39    end
40  end
41  if not newIterationRequired and scheduledIDRenamings not empty then
42    optimize ID renaming list;
43    traverse lattice domain and rename the IDs;
44  end
45 until not newIterationRequired ;

```

case, a *C-type message* is sent to all patches that know this third bubble. Also, if a process receives an A-type message, it has to notify other processes about changes on yet other bubbles, unknown to the first, by C-type messages in the next iteration. This situation can even cause a change of responsibility for a bubble merge, which may lead to a short-timed inconsistent state between the processes: For instance, if patch D merges bubbles 3 and 2 in Fig. 5, it sends a B-type message to C, which therefore knows that bubble 2 now merges with 1 on A, but still assumes to be responsible for merge of bubble 1 and 2. Line 11 in Alg. 1 ensures that in this case patch C issues no message to A, which would not expect one. The inconsistent state is resolved in the next iteration when patch A forwards the change via C-type message.

Depending on the scenario, several bubbles may be merged at the same time (e.g. bubble 11 can be merged to 5 while 10 is merged to 4 in Fig. 4). In order to avoid several cycles to rename bubble IDs in the cell-based data fields, renaming is only scheduled (line 23) and performed in the last iteration when all merges have been processed.

4.3 Efficiency of Merge Algorithm

Pohl implemented two different communication schemes for exchange of bubble data: One using all-to-all message tools from the MPI library, and one scheme where all processes are lined up on a chain and messages start at both ends, rippling the updates through the chain while every intermediate process adds its information (for more details see 24). Thus, either one global all-to-all communication step is needed, or $N - 1$ iterations with 2 local messages, if N is the process count.

In contrast, the merge algorithm introduced here, requires less iterations and communication, depending on the number of bubbles to merge and the number of processes involved. The very unlike scenario depicted in Fig. 4 is resolved after only 6 iterations of relatively local communications, compared to $N - 1 = 9$ iterations or one all-to-all message, respectively. For an equitable comparison, the exchange of bubble update data has to be included, which Pohl fused in the same step. But these messages again are relatively local communications. Figure 6 depicts a worst case scenario for the novel algorithm: If the 25 bubbles, occupying the whole domain, merge with each other in the same time step, the localized algorithm needs 25 iterations in addition to the preceding update step, which is more than the $N - 1 = 15$ iterations in case of the chain-like communication.

Whereas a fair comparison of the merge algorithms is problematic due to the dependency on the scenario, the outstanding advantage of the new method is the prevention of all-to-all communication for the bubble updates. Even if the new merge algorithm is less efficient, a merge (and even simple merges of two small bubbles) happens very seldom compared to the frequent volume change updates in every time step. Due to numerical limits of LBM, resolution commonly has to be chosen such that bubbles cover distances of one lattice cell in not less than 1 000 time steps.

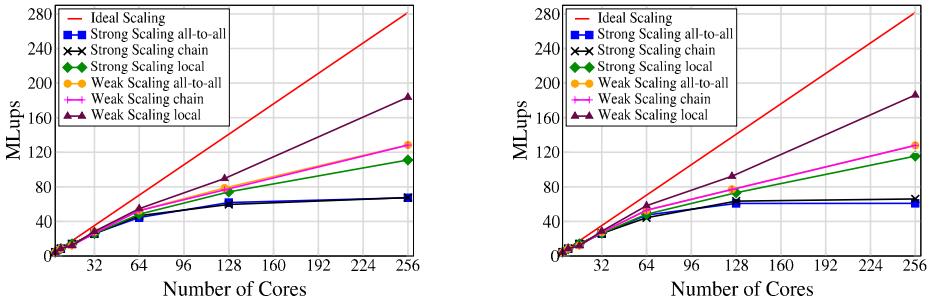


Fig. 7. Weak and strong scaling on a Woodcrest platform [32] simulating a single small rising bubble in the center of the domain (left) and two small rising bubbles far apart of each other (right). For strong scaling, the domain size is $320 \cdot 200 \cdot 200$ lattice cells (≈ 6.5 GB). Weak scaling scales from 199^3 (≈ 4 GB) on 4 cores to 794^3 (≈ 256 GB) on 256 cores.

5 Resulting Performance Gain

Performance was evaluated on two different architectures, an IA32-based cluster with two dual-core Woodcrest CPUs per node [32], and an IA64-based SGI Altix system, having a total count of 4864 dual-core Itanium 2 processors with non-uniform memory access (NUMA) to 39 TB of memory [33]. Measurement unit is *million lattice site updates per second* (MLups), which is a common unit for LBM enabling comparison of different implementations for the same model. The floating point operations (Flops) per Lup ratio depends strongly on interface cell count of the scenario. A liquid cell update is around 300 Flops, updating an interface cell costs approx. 3200 Flops. The scenario with one small bubble results in around 350 Flops/Lup (measured value). All performance experiments were performed with waLBerla, using one patch per process in order to avoid overhead due to local communication.

As Fig. 7 shows, the novel algorithm ensures better parallel performance and efficiency. It is obvious that a small bubble causes little communication effort for the local communication procedure, compared to the two all-to-all communication types. This is especially noticeable for strong scaling, because of two reasons: First, due to the fixed domain and bubble sizes, patches without contribution to simulation of the bubble appear earlier. Second, the communication-to-computation ratio increases and thus the effect of saved communication becomes more prominent. By the local algorithm, efficiency on 256 cores improved to 40% compared to 24%. Performance of simulating two bubbles that are apart of each other such that their treatment occurs independently—and thus in parallel—expectedly depicts the same scaling behavior (right graph in Fig. 7).

Whereas scaling efficiency is not affected by higher numbers of *independent* bubbles, numerous overlappings of bubbles and patches or simply larger bubbles increase the number of processes involved in the update of them and thus have an impact on performance. As a result, the performance gain of variant “local”

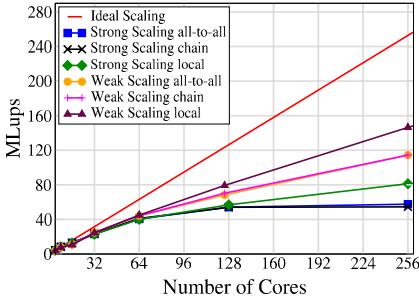


Fig. 8. Weak and strong scaling on a Woodcrest platform [32] simulating a larger rising bubble in the center of the domain. The diameter of the bubble is half of the domain’s width. Domain sizes like in Fig. 7.

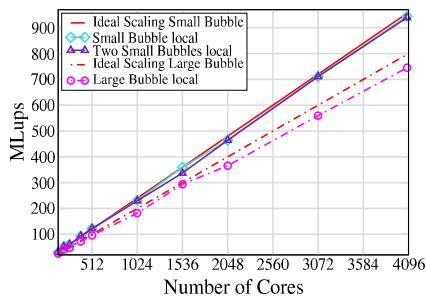


Fig. 9. Weak scaling on the Itanium 2-based Altix ‘HLRB2’ with Numalink4 interconnect [33]. The domain scales from 950^3 (≈ 439 GB) on 128 cores to 3010^3 (≈ 14 TB) on 4080 cores.

compared to the all-to-all communication schemes shrinks, as Fig. 8 shows for a simulation of a larger bubble, having a diameter of half the domain width.

The novel free-surface LBM implemented in waLBerla scales well also for large numbers of processes, as illustrated in Fig. 9, where a domain size of 3.5 GB per core was chosen to assess the potential in maximum simulation size. At the time, single-core performance on Itanium 2 is much less than on Woodcrest (0.35 MLups compared to 1.15 MLups). However, thanks to the good scaling and the huge memory resources of this computer, domain sizes of 3000^3 lattice cells and more are possible, enabling simulations of seriously challenging real-life engineering applications.

6 Conclusion

The parallel free-surface extension of the LBM exhibits massive computation and communication effort. Besides seven communication steps with direct neighbors, it involves an exchange of bubble volume update data in each time step, which has to be communicated among the processes that belong to the corresponding bubbles. In a previous implementation by Pohl [24] this is realized by all-to-all communication resulting in moderate parallel efficiency for more than hundred processors. The algorithm presented in this paper handles bubble exchange and coalescence locally and renders global communication unnecessary. Scaling comparison shows tremendous improvement of strong and weak scaling.

Acknowledgements. Special thanks to the cxHPC group of RRZE for their support and the computing privileges granted for the performance runs in this paper. This work is partially funded by the European Commission with *DECODE*, CORDIS project no. 213295, by the Bundesministerium für Bildung und

Forschung under the *SKALB* project, no. 01IH08003A, as well as by the “Kompetenznetzwerk für Technisch-Wissenschaftliches Hoch- und Höchstleistungsrechnen in Bayern” (*KONWIHR*) via *waLBerlaMC*.

References

1. Mashayek, F., Ashgriz, N.: A hybrid finite-element-volume-of-fluid method for simulating free surface flows and interfaces. *Int. J. Num. Meth. Fluids* 20, 1363–1380 (1995)
2. Ghidarsa, B.: Finite Volume-based Volume-of-Fluid method for the simulation of two-phase flows in small rectangular channels. PhD thesis, University of Karlsruhe (2003)
3. Gueyffier, D., Li, J., Nadim, A., Scardovelli, S., Zaleski, S.: Volume of fluid interface tracking with smoothed surface stress methods for three-dimensional flows. *J. Comp. Phys.* 152, 423–456 (1999)
4. Aulisa, E., Manservisi, S., Scardovelli, R., Zaleski, S.: A geometrical area-preserving volume-of-fluid method. *J. Comp. Phys.* 192, 355–364 (2003)
5. Chang, Y., Hou, T., Merriam, B., Osher, S.: A level set formulation of Eulerian interface capturing methods for incompressible fluid flows. *J. Comp. Phys.* 124, 449–464 (1996)
6. Sussman, M., Fatemi, E., Smereka, P., Osher, S.: An improved level set method for incompressible two-phase flows. *Comp. & Fl* 27(5–6), 663–680 (1998)
7. Osher, S., Fedkiw, R.P.: Level set methods: an overview and some recent results. *J. Comp. Phys.* 169(2), 463–502 (2001)
8. Zinchenko, A.Z., Rother, M.A., Davis, R.H.: Cusping, capture and breakup of interacting drops by a curvatureless boundary-integral algorithm. *J. Fluid Mech.* 391, 249 (1999)
9. Hou, T., Lowengrub, J., Shelley, M.: Boundary integral methods for multicomponent fluids and multiphase materials. *J. Comp. Phys.* 169, 302–362 (2001)
10. Gunstensen, A.K., Rothman, D.H., Zaleski, S., Zanetti, G.: Lattice Boltzmann model of immiscible fluids. *Phys. Rev. A* 43(8), 4320–4327 (1991)
11. Rothmann, D., Keller, J.: Immiscible cellular automaton fluids. *J. Stat. Phys.* 52, 1119–1127 (1988)
12. Tölke, J.: Gitter-Boltzmann-Verfahren zur Simulation von Zweiphasenströmungen. PhD thesis, Technical University of Munich (2001)
13. Shan, X., Chen, H.: Lattice Boltzmann model for simulating flows with multiple phases and components. *Phys. Rev. E* 47(3), 1815–1820 (1993)
14. Inamuro, T., Tomita, R., Ogino, F.: Lattice Boltzmann simulations of drop deformation and breakup in shear flows. *Int. J. Mod. Phys. B* 17, 21–26 (2003)
15. Orlandini, E., Swift, M.R., Yeomans, J.M.: A lattice Boltzmann model of binary-fluid mixtures. *Europhys. Lett.* 32, 463–468 (1995)
16. Swift, M.R., Orlandini, E., Osborn, W.R., Yeomans, J.M.: Lattice Boltzmann simulations of liquid-gas and binary fluid systems. *Phys. Rev. E* 54(5), 5041–5052 (1996)
17. He, X., Chen, S., Zhang, R.: A lattice Boltzmann scheme for incompressible multi-phase flow and its application in simulation of Rayleigh-Taylor instability. *J. Comp. Phys.* 152, 642–663 (1999)
18. Ginzburg, I., Steiner, K.: Lattice Boltzmann model for free-surface flow and its application to filling process in casting. *J. Comp. Phys.* 185, 61–99 (2003)

19. Körner, C., Thies, M., Hofmann, T., Thürey, N., Rüde, U.: Lattice Boltzmann model for free surface flow for modeling foaming. *J. Stat. Phys.* 121(1–2), 179–196 (2005)
20. Thürey, N.: Physically based Animation of Free Surface Flows with the Lattice Boltzmann Method. PhD thesis, Univ. of Erlangen (2007)
21. Thürey, N., Pohl, T., Rüde, U., Oechsner, M., Körner, C.: Optimization and stabilization of LBM free surface flow simulations using adaptive parameterization. *Comp. & Fl* 35(8–9), 934–939 (2006)
22. Thürey, N., Rüde, U.: Free surface lattice-Boltzmann fluid simulations with and without level sets. In: VMV, pp. 199–208. IOS Press, Amsterdam (2004)
23. Xing, X.Q., Butler, D.L., Ng, S.H., Wang, Z., Danyluk, S., Yang, C.: Simulation of droplet formation and coalescence using lattice Boltzmann-based single-phase model. *J. Coll. Int. Sci.* 311, 609–618 (2007)
24. Pohl, T.: High Performance Simulation of Free Surface Flows Using the Lattice Boltzmann Method. PhD thesis, Univ. of Erlangen (2008)
25. Pohl, T., Thürey, N., Deserno, F., Rüde, U., Lammers, P., Wellein, G., Zeiser, T.: Performance evaluation of parallel large-scale lattice Boltzmann applications on three supercomputing architectures. In: SC 2004: Proceedings of the 2004 ACM/IEEE conference on Supercomputing (2004)
26. Körner, C., Pohl, T., Rüde, U., Thürey, N., Zeiser, T.: Parallel lattice Boltzmann methods for CFD applications. In: Bruaset, A., Tveito, A. (eds.) *Numerical Solution of Partial Differential Equations on Parallel Computers. Lecture Notes for Computational Science and Engineering*, vol. 51, pp. 439–465. Springer, Heidelberg (2005)
27. Körner, C., Thies, M., Singer, R.F.: Modeling of metal foaming with lattice Boltzmann automata. *Adv. Eng. Mat.* 4, 765–769 (2002)
28. Succi, S.: The lattice Boltzmann equation for fluid dynamics and beyond. Oxford University Press, Oxford (2001)
29. Parker, B.J., Youngs, D.L.: Two and three dimensional Eulerian simulation of fluid flow with material interfaces. Technical Report 01/92, UK Atomic Weapons Establishment, Berkshire (1992)
30. Feichtinger, C., Götz, J., Donath, S., Iglberger, K., Rüde, U.: Concepts of waLBerla Prototype 0.1. Technical Report 07-10, Chair for System Simulation, Univ. of Erlangen (2007)
31. Feichtinger, C., Götz, J., Donath, S., Iglberger, K., Rüde, U.: WaLBerla: Exploiting massively parallel systems for lattice Boltzmann simulations. In: Trobec, R., Vajtersic, M., Zinterhof, P. (eds.) *Parallel Computing: Numerics, Applications, and Trends*. Springer, UK (2009)
32. Woody, <http://www.rrze.de/dienste/arbeiten-rechnen/hpc/systeme/woodcrest-cluster.shtml>
33. HLRB2, <http://www.lrz-muenchen.de/services/compute/hlrb/>

Fast Implicit Simulation of Oscillatory Flow in Human Abdominal Bifurcation Using a Schur Complement Preconditioner

K. Burckhardt¹, D. Szczerba^{1,2}, J. Brown³, K. Muralidhar⁴, and G. Székely¹

¹ Department of Electrical Engineering, ETH, Zürich, Switzerland

² IT'IS Foundation for Research, Zürich, Switzerland

³ Laboratory of Hydraulics, Hydrology and Glaciology, ETH, Zürich, Switzerland

⁴ Department of Mechanical Engineering, IIT Kanpur, India

Abstract. We evaluate a parallel Schur preconditioner for large systems of equations arising from a finite element discretization of the Navier-Stokes equations with streamline diffusion. The performance of the method is assessed on a biomedical problem involving oscillatory flow in a human abdominal bifurcation. Fast access to flow conditions in this location might support physicians in quicker decision making concerning potential interventions. We demonstrate scaling to 8 processors with more than 50% efficiency as well as a significant relaxation of memory requirements. We found an acceleration by up to a factor 9.5 compared to a direct sparse parallel solver at stopping criteria ensuring results similar to a validated reference solution.

Keywords: Aortic aneurysm, flow simulation, streamline diffusion FEM, indefinite matrix, parallel Krylov solver.

1 Introduction

Aortic aneurysm is a permanent and irreversible widening of the aorta, which, if left untreated, can dilate further and eventually rupture, leading to death in most cases. Many biochemical and biomechanical mechanisms have been identified as playing a role in the formation of aneurysms [12] [18]. Shear pattern and pressure distribution in the pathology are of considerable interest in this context, but computer tomography scans do not deliver such information and conventional magnetic resonance imaging (MRI) does not offer the appropriate accuracy. Numerical simulations offer therefore an attractive option to provide support for diagnosis and interventional planning. Computer aided quantifications, however, are still not part of clinical decision making. The major reason is the prohibitive computation time needed when using standard solution techniques, which makes this approach infeasible in an everyday clinical environment. Parallelization emerges therefore as an attractive technique to significantly shorten the simulation time in such critical cases.

We have already pointed out that flow disturbances due to oscillatory flow in a bifurcation is a possible factor in formation of aortic aneurysms [18]. Therefore,

knowledge of flow conditions in aneurysm prone regions might help to identify patients at risk. For flow investigations, a tetrahedral mesh of an abdominal bifurcation was extracted from MRI scans of a real patient (see Sect. 2.1). To find the pressure and velocity distribution in this domain, the Navier-Stokes equations are numerically solved. A mixed finite element discretization with streamline stabilization and oscillatory boundary conditions is used (Sect. 2.2). After implicit discretization in time, a nonlinear system of equations is obtained for each time step, the solution of which is computed iteratively by solving linear systems. The efficient solving of these linear systems, essential for quick patient diagnosis, is the topic of the present study.

Our fully implicit formulation circumvents the splitting error incurred by the semi-implicit projection methods frequently used for this problem [6]. Due to incompressibility, the fully implicit method relies on indefinite linear systems which are more expensive to solve than the definite systems arising in projection methods. Until recently, solution methods for indefinite systems have exhibited strongly mesh-dependent convergence rates in contrast to definite systems which can frequently be treated in a mesh-independent number of iterations using multi-level methods.

If high resolution is required, the bifurcation and other biomedical models generate equation systems with millions of degrees of freedom. Direct solvers are generally expensive in terms of storage and computing time [11] and their cluster parallelization poses many challenges. Iterative solvers are generally more suitable for large scale distributed memory computing, and a few toolkits are publicly available, e.g., Portable Extensible Toolkit for Scientific Computation (PETSc), [12]. However, they require preconditioning, but standard preconditioners fail as the linear systems in the fully implicit formulation are indefinite. In this study, we test the Schur complement preconditioner described in Sect. 3.2. It is embedded in a completely parallelized implementation of matrix assembly and solving (Sect. 4).

2 Numerical Representation of the Bifurcation Problem

2.1 Data Acquisition

An MRI scan of an abdominal bifurcation was acquired using a magnetic resonance system Philips Achieva 1.5T, using a 3-D phase contrast pulse sequence, which provided bright blood anatomy images of the lower abdomen of a healthy 35 year old male volunteer. Slices were taken perpendicularly to the infrarenal abdominal aorta. The images were 20 slices with 224 x 207 pixels in size, with a slice thickness of 5mm and an in-plane pixel spacing of 1.29 mm, over 24 phases of the cardiac cycle. The arterial lumen was then segmented from the anatomy images and smoothed, so as to produce an initial surface mesh. This was used as input to a volumetric meshing algorithm. A commercial meshing toll (ICEM CFD, see <http://www.ansys.com/products/icemcfd.asp>) implementing octree based refined subdivision has been used to produce high quality

tetrahedral meshes. The quality was defined as the normalized ratio of inscribed to circumscribed spheres radii and was 0.4 for the worst mesh element. For this study, we performed tests with three different mesh resolutions: 10'000 (mesh S), 50'000 (mesh M) and 150'000 (mesh L) second order tetrahedrons. Fig. 1 shows the middle sized mesh.

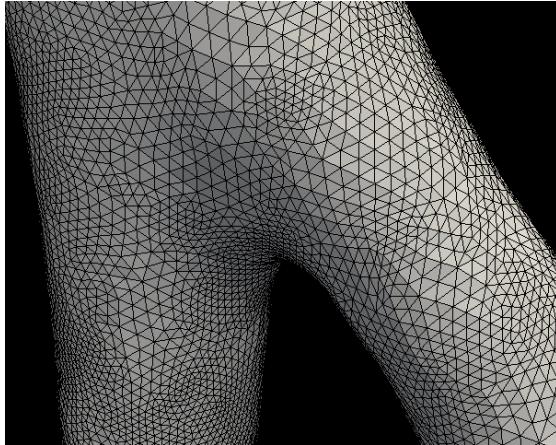


Fig. 1. Mesh

2.2 Discretization Scheme

Assuming constant viscosity η and density ρ , the Navier-Stokes equations for an incompressible fluid are given by

$$\rho \left(\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} \right) - \eta \nabla^2 \mathbf{u} + \nabla p = 0 \quad \text{and} \quad \nabla \cdot \mathbf{u} = 0,$$

where $\mathbf{u} = \mathbf{u}(x, y, z)$ denotes the velocity and $p = p(x, y, z)$ the pressure. To solve we apply a standard Galerkin finite element procedure [22] with streamline diffusion [5] for stabilization reasons. The time derivative is discretized using the backward Euler method, producing a nonlinear system of algebraic equations to be solved at each time-step. Picard iteration is applied to the nonlinear system, resulting in a linear system to be solved at each iteration. See Sect. 3 for details on this linear system. Picard iteration is preferable to Newton linearization insofar as the latter method can slow down the convergence of approximated Schur complement preconditioners by about a factor of two on the present problem [4].

The flow in abdominal aorta is known to be oscillatory [18], i.e., it includes a flow reversal phase, with peak inflow/outflow velocities of about $0.8/0.2\text{ms}^{-1}$, respectively. We approximate this condition by a sine wave multiplied by a developed flow profile on the inflow plane. On the outflow planes $p = 0$ and $d\mathbf{u}/d\mathbf{n}$ hold. Everywhere else on the bifurcation boundary the *no-slip* boundary condition is applied.

3 Solution Method

The linear systems in the Picard iteration have the block form

$$K\phi = \begin{pmatrix} A_u & B^T \\ B & 0 \end{pmatrix} \begin{pmatrix} \phi_u \\ \phi_p \end{pmatrix} = \begin{pmatrix} b_u \\ 0 \end{pmatrix}. \quad (1)$$

where ϕ_u, ϕ_p are velocity and pressure solution vectors respectively, and b_u is the body force. The matrix A_u represents the discretization of the time-dependent advection-diffusion equation and is non-symmetric in the presence of advection, and B is the discrete divergence operator.

Applying iterative techniques, a system such as (1) is solved without explicitly forming the inverse K^{-1} or factoring K . Hence, less storage and a smaller number of operations may be required than by direct methods. The essential algebraic operation is the matrix-vector product $y \leftarrow Kx$, which requires little computing time and is relatively unproblematic to implement on distributed memory systems [2], [3], [21].

The iterative methods of choice are Krylov subspace methods, for which the number of iterations required for convergence scales roughly with the square root of the condition number. Since the condition number of K scales as $1/h^2$ where h is the mesh size, preconditioning is mandatory for high-resolution. Due to the zero block, K is indefinite and standard preconditioners such as ILU, Schwarz, and multigrid perform poorly or fail completely.

3.1 Existing Methods

Preconditioners for the indefinite Navier-Stokes system are based on approximate factorizations of (1), with the Uzawa [19] and SIMPLE [15] algorithms corresponding to particular choices [8]. While these two classical methods and their variants such as SIMPLER [14] avoid the large memory requirements of direct methods, their performance is usually poor and they exhibit mesh-dependent convergence rates. More recently, several block preconditioners have been developed based on the factorization

$$\begin{pmatrix} A_u & B^T \\ B & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ BA_u^{-1} & 1 \end{pmatrix} \begin{pmatrix} A_u & 0 \\ 0 & S \end{pmatrix} \begin{pmatrix} 1 & A_u^{-1}B^T \\ 0 & 1 \end{pmatrix} \quad (2)$$

where $S = -BA_u^{-1}B^T$ is the Schur complement [4]. A linear solver for (1) is obtained by choosing an outer Krylov method, selectively dropping terms in (2), and choosing preconditioners for the definite matrices A_u and S . Modern choices which involve spectrally equivalent operators obtained using multigrid offer significant performance benefits in the form of mesh-independent convergence rates and low sensitivity to Reynold's number and time-step length. The matrix A_u is sparse and definite so it may be assembled and preconditioned using standard methods. Unfortunately S is dense, thus very expensive to form, so other methods must be employed to form its preconditioner.

3.2 Our Preconditioner

When right-preconditioned GMRES is applied to (II), an optimal preconditioner is

$$P^{-1} = \begin{pmatrix} A_u & B^T \\ 0 & S \end{pmatrix}^{-1}, \quad (3)$$

where $S = -BA_u^{-1}B^T$ is the Schur complement. Since KP^{-1} has minimum degree 2, the right-preconditioned system $KP^{-1}(Px) = b$ converges in two iterations of GMRES [13].

The matrix P^{-1} is not assembled, but only defined by its application to a vector

$$\begin{pmatrix} z_u \\ z_p \end{pmatrix} \leftarrow \begin{pmatrix} A_u & B^T \\ 0 & S \end{pmatrix}^{-1} \begin{pmatrix} y_u \\ y_p \end{pmatrix}. \quad (4)$$

The vector $(z_u, z_p)^T$ is obtained by solving the systems

$$Sz_p = y_p, \quad (5)$$

$$A_u z_u = y_u - B^T z_p. \quad (6)$$

Analogously to P , a matrix-free representation is also used for S . The action $r_p \leftarrow Sz_p$ of the Schur complement to a vector is obtained in three steps

$$\begin{aligned} \bar{z}_p &\leftarrow B^T z_p \\ \bar{r}_p &\leftarrow A_u^{-1} \bar{z}_p \\ r_p &\leftarrow -B \bar{r}_p. \end{aligned} \quad (7)$$

Using $y_u = b_u$ and $y_p = -BA_u^{-1}b_u$ and solving (7), (5), and (6) completely corresponds to the full factorization (2), hence solves (II) completely. Solving these inner systems to high precision is expensive. Only some iterations are usually performed, so the matrices in (3) are replaced by the approximations $\hat{S} \approx S$ and $\hat{A}_u \approx A_u$. If a Krylov method is used in the inner iterations, or if a variable number of stationary iterations are performed, the preconditioner is not a linear operation so the outer iteration must use a flexible variant such as FGMRES [16]. This solver allows only right preconditioning.

The action of applying P^{-1} to a vector requires solves with A_u in two places, one for computing the action of $-BA_u^{-1}B^T$ in (7) and one for computing z_u in (6). Since A_u is given in assembled form, any standard preconditioner for nonsymmetric problems can be used. Algebraic multigrid, e.g., was shown to be suitable for advection-diffusion problems including streamline-diffusion stabilization [10, p. 252].

3.3 Preconditioning the Schur Complement

A crucial point for the performance of P is effective preconditioning of $\hat{S}z_p = y_p$. Elman studied preconditioners for the Schur complement in the case of time-dependent Navier-Stokes equations [4]. He found experimentally that the so-called

BFBt method in conjunction with GMRES allows a convergence which is essentially independent of the Reynolds number and of the mesh size at finite time steps. Moreover, the iteration count was observed to decrease as a function of time step size. In the BFBt method, S is approximated by

$$P_S = (BB^T)(BA_uB^T)^{-1}(BB^T). \quad (8)$$

The application of P_S^{-1} requires two solves with BB^T at each iteration of this inner system. BB^T corresponds to a discrete Laplacian with appropriate boundary conditions, and according to Elman, an approximate solution by “one or two steps of V-cycle multigrid” is “sufficient for good performance of the complete solver” [4, p. 358].

4 Implementation

4.1 Mesh Partitioning

The total matrix is reassembled for each Picard iteration in each time step, which at a step size of 0.01 s for 4 cardiac cycles is about 2000 times. Computing time is hence a crucial factor, and the assembly is hence done in parallel. Each processor calculates roughly the same number of element stiffness matrices. At model setup, the master reads the mesh and distributes it to each processor. A small number of message start-ups at assembly is desirable to limit latency. Partitioning the mesh in stripes minimizes the number of neighbours of each sub-domain and hence the number of messages for exchanging the stashed entries of the interface nodes. PETSc offers an interface to the state-of-the-art partitioning software Chaco [9],

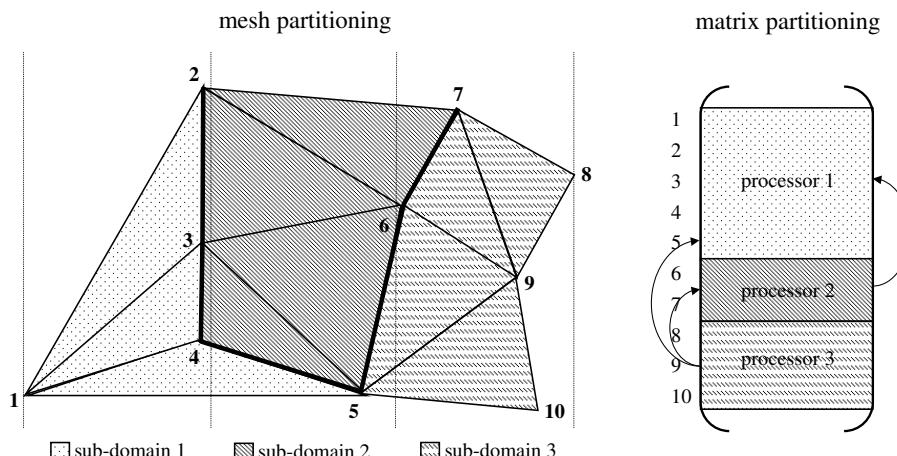


Fig. 2. Illustration of mesh and matrix partitioning. The arrows indicate the inter-processor communication during matrix assembly.

e.g., which contains a routine for cutting in stripes along an axis. However, the maximum number of subdomains is limited, and there is no option to keep the same axis when applying the routine recursively. We implemented therefore a simple partitioning method by cutting the mesh in equidistant stripes along its longest axis.

Each tetrahedron is assigned to the stripe where its center is located (see Fig. 2). They are re-ordered in the sense that the ones in the first stripe are send to processor one, the ones in the second to processor two, etc. The nodes are renumbered also according to the order of the sub-domains, meaning that the nodes of the elements of stripe one are listed at first, the ones of the elements of stripe two are listed afterwards, and so on. The interface nodes are always assigned to the processor with the smaller number. For example, the interface nodes between the sub-domains one and two belong to the first processor, and the ones between the sub-domains two and three to the second. The tetrahedral information of each sub-domain, i.e. lists of ten node indices per line, is translated according to the new node numbering and afterwards distributed.

4.2 Matrix Assembly

The global distributed matrices A_u , B , and B^T are assembled separately and in parallel using PETSc. Using this programming environment, matrices are partitioned in blocks of contiguous rows with a user-defined number of rows per processor. The assignment of the nodes described above defines the number of rows per processor. If N is the total number of nodes, A_u has the overall size $3N \times 3N$. Indicating with N^k the number of nodes of processor k , the stripe of A_u of this processor has the size $3N^k \times 3N$. The blocks B and B^T , however, have the overall size $N_P \times 3N$ and $3N \times N_P$, where N_P indicates the size of the pressure space. Accordingly, the number of rows of B^T per processor is the same as the one of A_u , whereas the local number of rows of B is given by the local number of pressure nodes.

Preallocation of the matrix memory is also important for a fast assembly. It requires the number of nonzeros in each row of A_u , B , and B^T , which are determined before starting the computation of the cardiac cycles. The (local) entries in the element stiffness matrices are added to the corresponding global entries in the overall matrices A_u , B , and B^T . If the global entry is in a locally owned row, no inter-processor communication is necessary. Otherwise the entry is stashed until the end of the assembly phase when PETSc sends all buffered values to the appropriate processors.

The global row or column index i_A of an entry in A_u is simply given by the index n of the corresponding node in the tetrahedral list, where $i_A = (n - 1) \cdot 3 + l$ and $l = 1, 2$, or 3 depending if it is a matter of the x , y , or z coordinate. Concerning the global row or column indices in B or B^T , respectively, it must be taken into account that $N_P \neq N$ in case of second order tetrahedra, e.g., and that the pressure node indices may not be consecutive. Therefore, the indices of all the pressure nodes are listed from minimum to maximum, and the global index i_B of a pressure node n is defined by the position of its index in this list.

Dirichlet boundary points fixing the velocity to certain values are eliminated from the equation system, which downsizes A_u to $3(N - N_B) \times 3(N - N_B)$, if N_B is the number of boundary points. The width of B and the height of B^T are reduced by the same amount, whereas the size of the other respective dimension of these matrices remains the same due to the absence of pressure constraints. Indicating with id_{free} the indices of the remaining unknowns, and with $a_{u\ ij}$, b_{ij} , and b_{ij}^T the elements of A_u , B , and B^T , respectively, the matrix blocks after the elimination are

$$\begin{aligned} A_u^* &= A_u(a_{u\ ij}), \quad \text{with } i, j \in \text{id}_{\text{free}}, \\ B^* &= B(b_{ij}), \quad \text{with } i \in [1, \dots, N_P], j \in \text{id}_{\text{free}}, \\ B^{*T} &= B^T(b_{ij}^T), \quad \text{with } i \in \text{id}_{\text{free}}, j \in [1, \dots, N_P]. \end{aligned}$$

These submatrices are extracted using the PETSc-function “MatGetSubMatrix”, which changes the matrix partitioning slightly but keeps the row and column order the same. The elimination leads to a change of the right hand side depending on the submatrices A_u^{tmp} and B^{tmp} , where

$$\begin{aligned} A_u^{\text{tmp}} &= A_u(a_{u\ ij}), \quad \text{with } i \in \text{id}_{\text{free}}, j \in \text{id}_{\text{fix}} \\ B^{\text{tmp}} &= B(b_{ij}), \quad \text{with } i \in [1, \dots, N_P], j \in \text{id}_{\text{fix}}. \end{aligned}$$

The symbol id_{fix} indicates the indices of the fixed values. The new overall right hand side b^* is given by

$$b^* = \begin{pmatrix} b_u \\ 0 \end{pmatrix} - \begin{pmatrix} A_u^{\text{tmp}} \\ B^{\text{tmp}} \end{pmatrix} u_{\text{fix}},$$

where the vector u_{fix} contains the boundary values.

4.3 Solver Set-Up

The equation system finally solved is

$$K^* \begin{pmatrix} \phi_u^* \\ \phi_p \end{pmatrix} = \begin{pmatrix} A_u^* & B^{*T} \\ B^* & 0 \end{pmatrix} \begin{pmatrix} \phi_u^* \\ \phi_p \end{pmatrix} = b^*. \quad (9)$$

The velocity solution vector ϕ_u^* has the size of id_{free} . It is filled-up with the boundary values after each solve. As mentioned above, (9) is solved using FGMRES and the preconditioner

$$P = \begin{pmatrix} \hat{A}_u^* & B^{*T} \\ 0 & \hat{S}^* \end{pmatrix},$$

where the stars indicate that the matrix blocks with eliminated constraints are inserted in (6) and (7). A relative tolerance of $\text{rtol} = 10^{-8}$ is chosen as stopping criteria, leading to a maximum deviation from the direct solution of 0.004%. The solution of the previous solve is used as initial guess.

The inner systems are generally solved using GMRES and left preconditioning. The parallel algebraic multigrid preconditioner boomerAMG [20] is used for (6) and (7) as well as for the solve with $B^* B^{*T}$ when preconditioning (5). The initial solution is always set to zero. The stopping criteria are optimized in terms of computing time. They are defined a priori either by fixing the iteration counts or by applying only the preconditioner. In the latter case, the preconditioning matrix is used as approximation for A_u^{*-1} , e.g., and applied to the right hand side without invoking GMRES.

The optimal solver parameters were found empirically using a script. One time step was simulated repeatedly while testing different combinations of solver options. Applying only the preconditioner resulted the fastest option for the solves with $B^* B^{*T}$ and with A_u^* in (7) at all mesh sizes. This was also true for system (6) in case of the meshes M and L, but a better performance could be achieved with 4 iterations of GMRES in case of the smallest mesh. The optimal iteration count when solving the Schur complement system was found to be 2 and 3 in case of mesh S and of the meshes M and L, respectively.

5 Results

The simulation was run on an SMP shared memory computer with 8 Intel/Xeon CPUs and on the CRAY XT3 computer system at the Swiss National Supercomputing Centre-CSCS consisting of 1664 dual-core computing nodes. Four cardiac cycles were computed at time steps of 0.01 s and with an overall time of 3 s. In Fig. 3, the velocity during flow reversal phase using mesh M is plotted as example. For comparison, the simulation was also run on Xeon by solving the overall equation system (II) using the shared-memory sparse direct solver Pardiso [17].

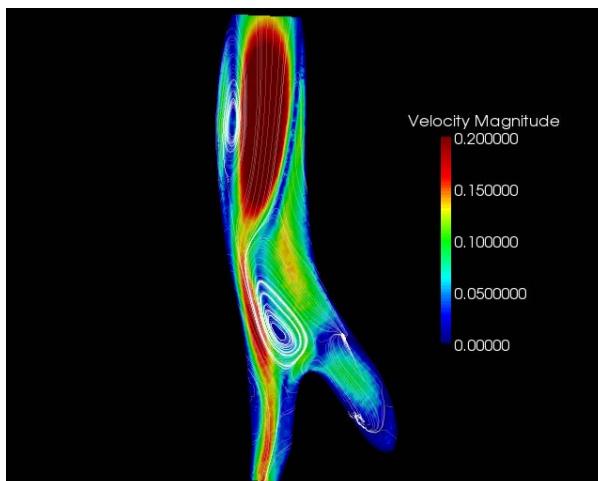


Fig. 3. Highly vortical flow during flow reversal phase, simulated using mesh M

5.1 Computing Time at Different Mesh Sizes

The method under study resulted up to 9.5 times faster than the direct solver (see Tab. 1). The Schur complement approach is less sensitive to the number of degrees of freedom. The increase in simulation time is due to the greater number of operations and the more extensive memory access at a greater amount of data but not due to a greater iteration count. The mean number of iterations in all the solves is about the same in mesh M and L, which confirms the finding in [4] that the convergence is almost independent from the mesh size using the BFBt preconditioner for the Schur complement. In mesh S, the average iteration count is smaller, but this is only due to the different solver parameters (see 4.3).

Table 1. Total assembly and solving time of 4 cardiac cycles at different mesh sizes, computed using 8 processors

mesh	method	computer	wall clock time [h]	mean iteration count
				per solve
S	direct	Xeon	4.56	—
	Schur Prec.	Xeon	0.80	4.4
	Schur Prec.	Cray	0.63	4.4
M	direct	Xeon	52.22	—
	Schur Prec.	Xeon	8.30	9.1
	Schur Prec.	Cray	4.98	9.1
L	direct	Xeon	331.43	—
	Schur Prec.	Xeon	35.07	8.2
	Schur Prec.	Cray	22.24	8.2

5.2 Scaling on the Shared Memory System

Fig. 4a shows the speed-up on Xeon when running the simulation with mesh S. An ideal scalability is most certainly prohibited by the fact that the speed of sparse matrix computations is also determined by the speed of the memory, not only the speed of the CPU.

5.3 Scaling on the Distributed Memory System

Fig. 4b shows the scaling on the Cray when running the simulation with mesh S and mesh L. First, a speed-up can be observed, but at a higher number of processors the performance degrades. This is due to the communication costs which exceed the speed gained by the parallelization at smaller problem sizes. According to [1], a minimum of 10'000 unknowns per processor is required to overweight the communication time. This number is reached in mesh S and in mesh L at about 5 and 70 processors, respectively. Generally, the simulation is faster on the Cray than on the Xeon although the CPUs of both computers have the same speed (2.6 GHz). Using 8 processors, mesh L, e.g., takes only about 22 h, instead of 35 h on Xeon (see Tab. 1). This is most likely due to the fact that memory access per CPU and communication are faster on the Cray.

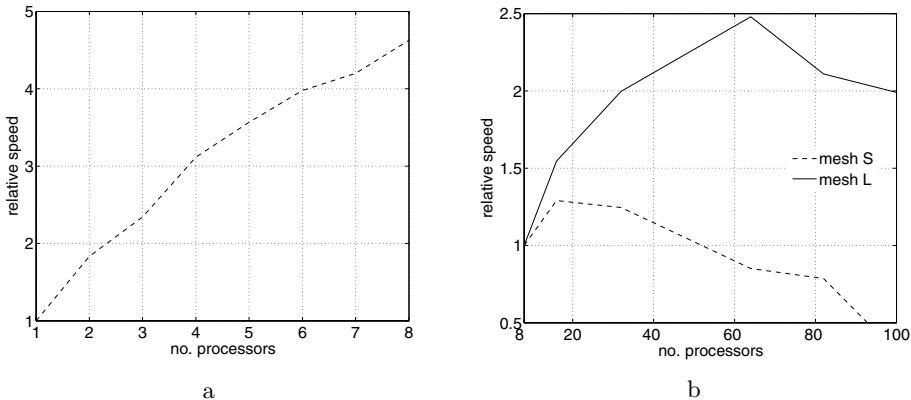


Fig. 4. The speed-up on Xeon (a) and on Cray (b) versus the number of processors

6 Discussion

We have presented a method to speed up simulations of oscillatory flow in a human abdominal bifurcation. Its main advantage consists in the independency of the convergence rate from the domain size or the resolution. Moreover, the method runs on distributed memory computers allowing meshes and equation systems which exceed significantly the storage capacity of shared-memory machines. The speed-up, however, on the distributed-memory systems was found to be limited at higher numbers of processors by the communication costs overweighting the gain in speed by the parallelization. This problem is inherent to parallel iterative solvers. Still, it is possible to reduce the computation time from about 14 days when using a parallel direct solver on a shared-memory system with 8 processors to 9 h using 64 processors on a Cray XT3. On the shared-memory system, an acceleration by up to a factor 9.5 compared to the direct solve has been observed.

The simulation could be accelerated by a less strict stopping criteria. In this study, the iteration was conducted until a relative tolerance of $rtol = 10^{-8}$ implying the small deviation from the direct solution of at most 0.004%. Moreover, only a simple version of the BFBt preconditioner for the Schur complement was implemented. A speed-up could probably be achieved by using the scaled version as proposed in [7], where a reduction of the iteration counts to approximately 75% is reported at Reynolds numbers comparable to the one in this simulation.

The method could be improved further by optimizing the partitioning of mesh and matrix. Ideally, each processor possesses the same number of matrix rows or unknowns. This number is defined in our implementation by the mesh partitioning. Up to now, the mesh is divided in equidistant stripes, and the number of nodes or unknowns per processor varies in mesh L at 8 processors, e.g., between 16197 and 43916. Taking the anisotropic node density into account by choosing the thickness so that each stripe contains the same number of nodes would lead to an even distribution of the matrix and so to the same amount of work for each processor. An acceleration of roughly a factor 1.5 can be expected. Additionaling the above

mentioned improvements, a total further speed-up by at least a factor 2 should be feasible.

Acknowledgement

This work was supported by a grant from the Swiss National Supercomputing Centre-CSCS.

References

1. Balay, S., Buschelman, K., Gropp, W.D., Kaushik, D., Knepley, M.G., Curfman McInnes, L., Smith, B.F., Zhang, H.: PETSc Web page (2001),
<http://www.mcs.anl.gov/petsc>
2. Demmel, J.W., Heath, M.T., van der Vorst, H.A.: Parallel numerical linear algebra. *Acta Numer.* 2, 111–197 (1993)
3. Dongarra, J.J., Duff, I.S., Sorensen, D.C., van der Vorst, H.A.: Numerical Linear Algebra for High-Performance Computers. SIAM Press, Philadelphia (1998)
4. Elman, H.C.: Preconditioning strategies for models of incompressible flow. *Journal of Scientific Computing* 25, 347–366 (2005)
5. Ferziger, J.H., Tseng, Y.H.: A coordinate system independent streamwise upwind method for fluid flow computation. *International Journal for Numerical Methods in Fluids* 45, 1235–1247 (2004)
6. Guermond, J.L., Minev, P., Shen, J.: An overview of projection methods for incompressible flows. *Computer Methods in Applied Mechanics and Engineering* 195, 6011–6045 (2006)
7. Elman, H., Howle, V., Shadid, J., Shuttleworth, R., Tuminaro, R.: Block preconditioners based on approximate commutators. *SIAM J. Sci. Comput.* 27(5), 1651–1668 (2006)
8. Elman, H., Howle, V., Shadid, J., Shuttleworth, R., Tuminaro, R.: A taxonomy and comparison of parallel block multi-level preconditioners for the incompressible navier-stokes equations. *Journal of Computational Physics* 227(1), 1790–1808 (2008)
9. Hendrickson, B., Leland, R.: The chaco user's guide: Version 2.0. Technical Report SAND 94–2692, Sandia (1994)
10. Kay, D., Loghin, D., Wathen, A.: A preconditioner for the steady-state navier-stokes equations. *SIAM J. Sci. Comput.* 24(1), 237–256 (2002)
11. May, D.A., Moresi, L.: Preconditioned iterative methods for stokes flow problems arising in computational geodynamics. *Physics of Earth and Planetary Interiors* 171, 33–47 (2008)
12. McGregor, R.H.P., Szczerba, D., Székely, G.: A multiphysics simulation of a healthy and a diseased abdominal aorta. In: Ayache, N., Ourselin, S., Maeder, A. (eds.) MIC-CAI 2007, Part II. LNCS, vol. 4792, pp. 227–234. Springer, Heidelberg (2007)
13. Murphy, M.F., Golub, G.H., Wathen, A.J.: A note on preconditioning for indefinite linear systems. *SIAM J. Sci. Comput.* 21(6), 1969–1972 (2000)
14. Patankar, S.V.: Numerical Heat Transfer and Fluid Flow. McGraw-Hill, New York (1980)
15. Patankar, S.V., Spalding, D.B.: A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows. *Int. J. Heat Mass Transfer* 15, 1787–1806 (1972)

16. Saad, Y.: A flexible inner-outer preconditioned gmres. *SIAM J. Sci. Comput.* 14(2), 461–469 (1993)
17. Schenk, O., Gärtner, K.: Solving unsymmetric sparse systems of linear equations with pardiso. *Journal of Future Generation Computer Systems* 20(3), 475–487 (2002)
18. Szczerba, D., McGregor, R., Muralidhar, K., Székely, G.: Mechanism and localization of wall failure during abdominal aortic aneurysm formation. *Biomedical Simulation* 5104, 119–126 (2008)
19. Uzawa, H.: Iterative methods for concave programming. In: Arrow, K.J., Hurwicz, L., Uzawa, H. (eds.) *Studies in Linear and Nonlinear Programming*, pp. 154–165 (1958)
20. Van Emden, H., Meier Yang, U.: Boomeramg: A parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics* 41, 155–177 (2002)
21. Wesley, P., Arbenz, P.: *Introduction to Parallel Computing*. Oxford University Press, Oxford (2004)
22. Zienkiewicz, O.C., Taylor, R.L.: *The Finite Element Method*, 5th edn. Butterworth-Heinemann, Butterworth (2000)

A Parallel Rigid Body Dynamics Algorithm

Klaus Iglberger and Ulrich Rüde

Friedrich-Alexander University Erlangen-Nuremberg,
91058 Erlangen, Germany

klaus.iglberger@informatik.uni-erlangen.de
<http://www10.informatik.uni-erlangen.de/~klaus/>

Abstract. For decades, rigid body dynamics has been used in several active research fields to simulate the behavior of completely undeformable, rigid bodies. Due to the focus of the simulations to either high physical accuracy or real time environments, the state-of-the-art algorithms cannot be used in excess of several thousand rigid bodies. Either the complexity of the algorithms would result in infeasible runtimes, or the simulation could no longer satisfy the real time aspects.

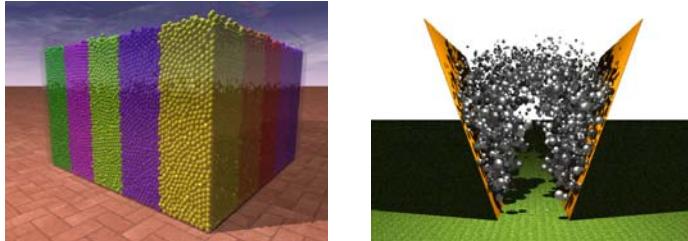
In this paper we present a novel approach for large-scale rigid body dynamics simulations. The presented algorithm enables for the first time rigid body simulations of several million rigid bodies. We describe in detail the parallel rigid body algorithm and its necessary extensions for a large-scale MPI parallelization and show some results by means of a particular simulation scenario.

1 Introduction

The field of rigid body dynamics is a subfield of the classical mechanics introduced by Isaac Newton in the 17th century. The only assumption made in this field is that the treated bodies are completely rigid and cannot be deformed by external forces. Though the rigidity assumption is a rigorous idealization of real materials, many phenomena in nature and problems posed in engineering can be well approximated in this way.

For instance, rigid body simulation has been used for decades in the robotics community that is primarily interested in the accurate prediction of friction forces. In material science, rigid body simulation is used for granular media simulations (see for example Fig. 1(a)) and in the computer graphics, computer games and virtual reality communities rigid body dynamics is used for authentic physical environments. For some years, rigid body dynamics has also been used in combination with lattice Boltzmann flow simulations to simulate the behavior of rigid bodies in a flow [10]. This coupled simulation system is for example used to simulate fluidization processes with a high number of buoyant rigid bodies (see for example Fig. 1(b)).

The most obvious difference in the existing rigid body simulation frameworks is the applied solver for the treatment of collisions between rigid bodies. Currently, the available algorithms for the calculation of collision responses can be



(a) Medium-sized simulation of a granular media scenario consisting of 175 000 spheres. The colors indicate the domains of the 25 involved MPI processes.

(b) Coupled simulation between a lattice Boltzmann flow simulation and the *pe* rigid body physics engine.

Fig. 1. Simulation examples for rigid body simulations

very coarsely subdivided into two categories: the accurate formulations based on linear complementarity problems (LCP) that are able to accurately predict frictional contact forces at higher computational costs [5,15,11,4,14], and fast algorithms that scale linearly with the number of contacts but suffer from a reduced accuracy [13,6].

Due to the computational costs of LCP solvers (for instance extensions of the Gauss-Seidel, Gauss-Jacobi or Conjugate Gradient methods), the maximum number of rigid bodies in such a simulation is limited to several thousand rigid bodies. A larger number of rigid bodies would result in completely infeasible runtimes. The faster algorithms, on the other hand, can handle more rigid bodies due to their linear complexity, but are often also limited to several thousand to ten thousand rigid bodies because of their real time requirements.

The algorithm presented in this paper represents the first large-scale rigid body dynamics algorithm. It is based on the fast frictional dynamics algorithm proposed by Kaufman et al. [12] and is parallelized with MPI [9]. Using this algorithm, our rigid body simulation framework named *pe* (the abbreviation for physics engine) enables the simulation of several million interacting rigid bodies on an arbitrary number of cores. As a first example of a large-scale rigid body simulation, the example demonstrated in Fig. 2 shows the simulation of 500 000 spheres and boxes falling into a well built of 3 000 fixed boxes. The simulation domain is partitioned into 91 subdomains, where each subdomain is managed by a single MPI process. All bodies contained in a subdomain are exclusively known to this managing process. Therefore all processes have to communicate with neighboring processes about rigid bodies crossing a process boundary. Due to the hexagonal shape of each subdomain, each process has to communicate with a maximum number of six neighboring processes. Please note that this simulation only rudimentarily demonstrates the full capacity of the parallel algorithm and was chosen such that individual objects are still distinguishable in the visualization.

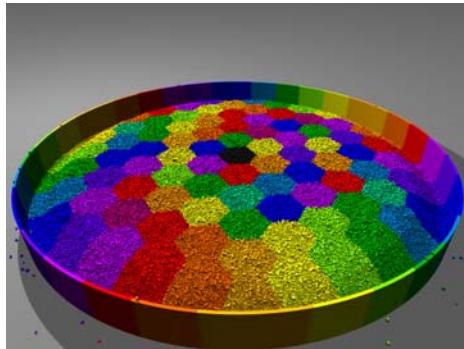


Fig. 2. Simulation of 500 000 spheres and boxes falling into a well built of 3 000 fixed boxes. The colors indicate the domains of the 91 MPI processes. Due to the hexagonal setup of the domains, each MPI process has a maximum number of six neighboring processes.

This paper is structured in the following sections: Sec. 2 gives an overview of related work in the field of rigid body dynamics and large-scale simulations. In Sec. 3, the focus lies on the parallel rigid body algorithm. This section explains the MPI parallelization of the algorithm and will explain the extensions of the original fast frictional dynamics algorithm. Sec. 4 analyzes of the scaling behavior of the parallel algorithm, whereas Sec. 5 concludes the paper.

2 Related Work

Tasora et al. [16] presented a large-scale parallel multi-body dynamics algorithm for graphical processing units. Their algorithm is based on a cone complementarity problem for the simulation of frictional contact dynamics. Their approach is suited for more than one million rigid bodies. There exist several alternative rigid body simulation frameworks that basically offer the same rigid body functionalities as our framework. Two open source examples are the Open Dynamics Engine (ODE) [1] and the OpenTissue library [2]. However, currently none of them offers massively MPI parallel rigid body dynamics as introduced in this paper.

Fleissner et al. [8] focus on parallel simulation of particle-based discretization methods, such as the discrete element method or smoothed particle hydrodynamics, instead of rigid body dynamics. Although their approach offers the option to run large-scale particle simulations, they are only considering point masses instead of fully resolved rigid bodies.

3 The Parallel Rigid Body Algorithm

The parallel rigid body dynamics algorithm presented in this section is based on the fast frictional dynamics (FFD) solver first published by Kaufman et al.

[2] and improved by Wengenroth [17]. Due to its formulation, this algorithm belongs to the group of fast, real time collision response algorithms. For a detailed explanation of the nonparallel version of this algorithm, please refer to either one of the two previously mentioned references. This section will primarily focus on the parallelization of this algorithm.

The FFD solver is particularly suited for a large-scale parallelization due to its strictly local collision treatment: in order to calculate a post-collision velocity for a colliding rigid body, only the states of the contacting rigid bodies are considered. Therefore it is not necessary to set up and solve a LCP in parallel as it would be necessary in case of the LCP-based collision response algorithm. Additionally, the complexity of the FFD algorithm is linearly depending on the number of rigid bodies and the number of contact points between these bodies.

Alg. II shows the parallel FFD algorithm. In contrast to the nonparallel algorithm, the parallel version contains a total of four MPI communication steps to handle the distributed computation (see the lines 1, 6, 21 and 34). The rest of the algorithm remains unchanged in comparison to the nonparallel formulation.

Instead of immediately starting with the first position and velocity half-step for every rigid body, the first step in every time step of the parallel algorithm is the synchronization of the external forces (Alg. II line 1). In case a rigid body is overlapping a process boundary, part of the external forces can be contributed by the remote processes. This scenario happens for instance in case the rigid body is immersed in a fluid flow (as illustrated in Fig. 1(b)). Part of the hydrodynamic forces are calculated by the local process, another part by the remote processes. Therefore the involved processes have to synchronize their forces and calculate a total force for every rigid body.

After the force synchronization, all local rigid bodies (i.e. all rigid bodies whose reference point is contained in the local process) are updated with the first position half-step and first velocity half-step. Note that only local rigid bodies are treated. Every rigid body is updated by exactly one process, i.e. by the process its center of mass is contained in. Remote rigid bodies (i.e. all rigid bodies whose reference points are not contained in the local process) are updated in the second MPI communication step (Alg. II, line 6). This communication step involves a synchronization of the position, the orientation and the velocities of remote rigid bodies, where the process that contains the reference point of the body sends the updated values to all neighboring remote processes that only contain part of the body. Due to the position and orientation change of all rigid bodies, it may also be the case that a rigid body now newly overlaps a process boundary to a particular neighboring remote process. In this case the entire rigid body has to be sent to the remote process, additionally including information about its geometry (as for instance the radius of a sphere, the side lengths of a box, the triangle mesh of an arbitrary geometry) and its material (containing information about its density, the coefficient of restitution and frictional parameters). With this information, the remote process is now able to create a copy of the rigid body.

Algorithm 1. The Parallel FFD-Algorithm

```

1 MPI communication step 1: force synchronization
2 for each body  $B$  do
3   position half-step ( $\phi_B^+$ )
4    $\phi_B^-$  = velocity half-step ( $B$ )
5 end
6 MPI communication step 2: update of remote and notification of new rigid
bodies
7 for each body  $B$  do
8   find all contacts  $\mathbf{C}(B)$ 
9   for each contact  $k \in \mathbf{C}(B)$  do
10    calculate twist  $\mathbf{n}_k$  induced by contact normal
11    calculate constraint offset  $d_k$ 
12    if constraint is violated ( $B, \phi_B^-, \mathbf{n}_k, d_k$ ) then
13      add collision constraint  $\mathbf{n}_k, d_k$  to  $\mathbf{T}(B)$ 
14      for  $m = 1 \dots \text{SAMPLESIZE}$  do
15        calculate twist  $\mathbf{s}_m$  induced by  $m^{\text{th}}$  sample  $\perp$  to contact normal
16        add friction constraint  $\mathbf{s}_m, \mu_m$  to  $\mathbf{S}(B)$ 
17      end
18    end
19  end
20 end
21 MPI communication step 3: exchanging constraints on the rigid bodies
22 for each body  $B$  do
23  if  $B$  has violated constraints then
24    find post-collision velocity  $\phi_B^\tau$  on the border of  $\mathbf{T}(B)$  closest to  $\phi_B^-$ 
25     $\mathbf{r}_B = \phi_B^- - \phi_B^\tau$ 
26    select friction response  $\delta_B$  from  $\mathbf{S}(B)$  minimizing  $\delta_B + \phi_B^\tau$ 
27     $\phi_B^+ = \phi_B^\tau + \epsilon \cdot \mathbf{r}_B + \delta_B$ 
28  end
29  else
30     $\phi_B^+ = \text{velocity half-step } (B)$ 
31  end
32  position half-step ( $B$ )
33 end
34 MPI communication step 4: update of remote and notification of new rigid
bodies

```

Note that, due to efficiency reasons, updates should be preferred to sending the complete rigid body. Only in case the remote process does not know the rigid body, sending the full set of information cannot be avoided. Also note, that it may be necessary to send updates for a rigid body that is no longer overlapping a particular boundary in order to give the remote process the information that the rigid body has left the domain of the remote process and can therefore be destroyed on the remote process.

The second communication step is followed by the collision detection that creates contact points for every pair of rigid bodies that is in contact. Directly afterwards,

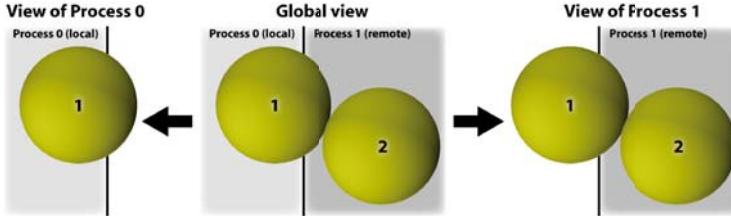


Fig. 3. Some collisions can only be detected on remote processes. In the illustrated example, process 0 only knows the local sphere 1, for which collision constraints have to be calculated. Process 1, on the other hand, knows both sphere 1 and 2 (where sphere 1 is remote and 2 is local). Therefore only process 1 can setup the constraints for the collision between spheres 1 and 2 and has to send these constraints to process 0, where they are used to handle the collision between the two spheres.

the setup of all collision constraints can be performed (see Alg. 1, line 7-20): for every contact point attached to the rigid body, a single collision constraint and several friction constraints are created in case the collision constraint is violated.

In the consecutive third communication step (Alg. 1, line 21), the MPI processes exchange collision and friction constraints among each other such that every process knows all active constraints on all local rigid bodies. This exchange is necessary since some constraints can only be detected on remote processes, as illustrated in Fig. 3. These constraints can now be used to treat all collisions and to update the velocities of the colliding rigid bodies. In case a rigid body is not involved in a collision, the velocity is updated in a second velocity half-step. After the velocity update (either due to collision handling or a velocity half-step), the second position half step is performed. In order to cope with this second change of the position and orientation of the rigid bodies, the fourth communication step (Alg. 1, line 34) performs the same synchronization as the second communication step. After this communication, all rigid bodies on all processes are synchronized and in a consistent state and the time step is finished.

For every communication step during the parallel rigid body simulation, the send buffers for the neighboring processes have to be filled individually with the according informations. This encoding step is succeeded by the message exchange via MPI, which is then followed by the decoding phase of the received messages. The complete MPI communication of the *pe* engine is shown in Alg. 2. The first step during communication is the initialization of the transfer of the according byte-encoded messages to the neighboring MPI processes via the non-blocking message passing function `MPI_Isend()`. While the MPI system handles the send operations, the receive operations are initiated. Since the size of the messages is unknown, the `MPI_Probe()` and `MPI_Get_count()` functions are used to test any incoming message. After that, the size of the receive buffer is adjusted and the blocking message passing function `MPI_Recv()` is used to receive the message. After all messages have been received, it is necessary to wait for all non-blocking send operations to finish. Only then the send buffers can be cleared for the next communication step.

Algorithm 2. MPI communication

```

1 for each neighboring process do
2   Encode the message to the neighboring process
3   Start a non-blocking send operation of the according byte encoded message
   via MPI_Isend()
4 end
5 for  $i$  from 0 to the number of neighboring processes-1 do
6   Check for any incoming message via MPI_Probe()
7   Acquire the total size of the message via MPI_Get_count()
8   Adjust the buffer size of the receiving buffer
9   Receive the message via MPI_Recv()
10  Decode the received message
11 end
12 Wait until all non-blocking send operations are completed
13 Clear the send buffers

```

Fig. 4 gives an example of a single moving rigid body that crosses the boundary between two processes. The example explains in detail, when the rigid body has to be sent by which process and when the responsibility for a rigid body is transferred to another process.

4 Results

In this section, the scaling behavior of the parallel FFD algorithm is analyzed closely. The machine we are using for our scaling experiments is the Woodcrest cluster at the regional computing center Erlangen (RRZE) [3]:

- 217 compute nodes, each with two Xeon 5160 Woodcrest chips (4 cores) running at 3.0 GHz, 4 MB Shared Level 2 Cache per dual core, and 8 GB of RAM
- Infiniband interconnect fabric with 10 GBit/s bandwidth per link and direction
- Overall peak performance of 10.4 TFlop/s (LINPACK result: 6.62 TFlop/s)

We are using up to 512 cores in order to simulate up to 8 million rigid bodies in a single simulation. All executables are compiled with the Intel 11.0 compiler (using the optimization flag `-O3`) and we are using Intel-MPI in the version 3.1.038.

The scenario we have chosen for our scaling experiments is illustrated in Fig. 5: we are simulating the movement of spherical particles in an evacuated box-shaped volume without external forces. For this we create a specified number of uniformly distributed spheres, each with a random initial velocity. This scenario is well suited for our scaling experiments, since it guarantees an equally distributed load even without load balancing strategies and similar communication costs on all involved processes.

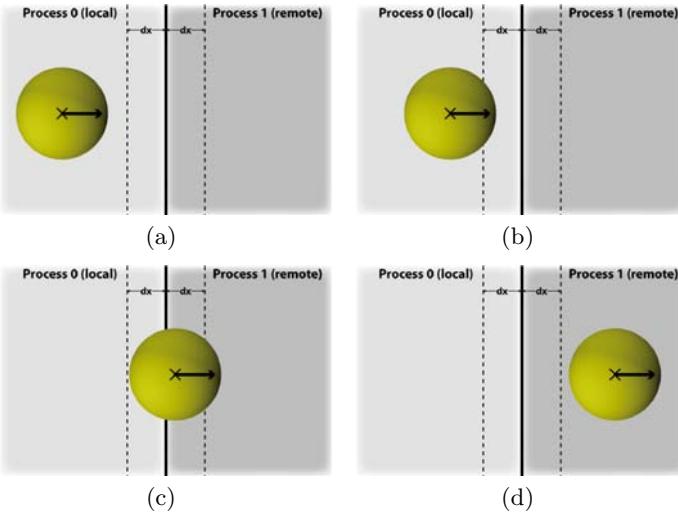


Fig. 4. Example of the communication of a single rigid body. The reference point of the rigid body is indicated by the cross in its center, its linear velocity by the arrow. In Fig. 4(a), the rigid body is solely contained in process 0. Process 1 does not know the rigid body. In Fig. 4(b), the rigid body has moved into the overlap region between the two processes and therefore has to be sent to process 1. Only in the first communication, the entire rigid body, including information about its geometry and its material have to be sent. In subsequent communications, it is sufficient to send updates for the body. In Fig. 4(c), the rigid body's reference point is no longer contained in process 0, but has moved to process 1. Therefore process 1 now considers the rigid body as local, process 0 only as remote. In Fig. 4(d), the rigid body has left the overlap region between the two processes and therefore doesn't have to be communicated anymore. Therefore process 0 no longer contains any part of the body, so the body is locally destroyed. Note that the choice of the size of dx is arbitrary, but larger than the minimum distance between two colliding bodies.

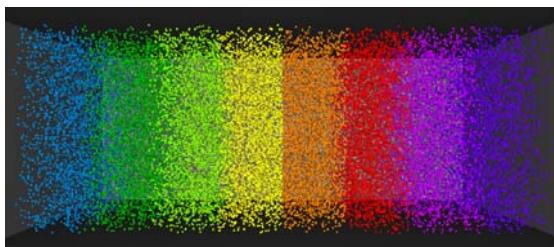


Fig. 5. Simulation scenario for the scaling experiments: the specified number of spherical rigid bodies are moving randomly in a closed, evacuated, box-shaped domain without external forces.

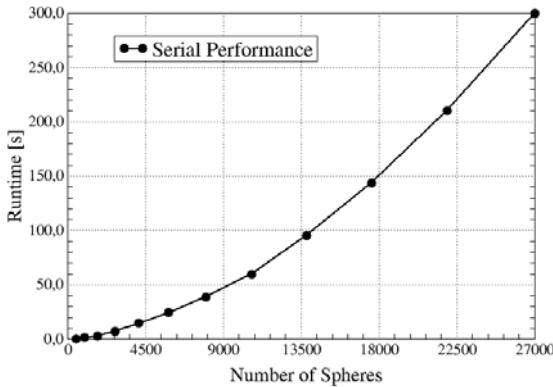


Fig. 6. Serial performance of 1000 time steps in the scenario for the strong and weak scaling experiments

In order to be able to analyze the parallel performance of the algorithm, it is first necessary to investigate the serial performance of the rigid body framework. Fig. 6 shows a serial scaling experiment with up to 27000 spheres in the scenario described above. Apparently the rigid body simulation exhibits a scaling behavior nonlinear with the number of bodies. One of the main reasons for this behavior is the lack of perfectly linear algorithms for some phases of the algorithm, as for instance the contact detection phase. In our simulation, we are using the sweep-and-prune algorithm as described in [7] for the coarse collision detection. Although this is one of the fastest available algorithms, it doesn't offer a perfectly linear scaling behavior.

Table I contains the results for a strong scaling experiment with 65 536 spheres initially arranged in a $64 \times 64 \times 16$ grid. In this experiment, the spheres use approximately 1% of the total volume of the domain. The first observation is that doubling the number of CPU cores improves the performance of the simulation beyond a factor of two. This can easily be explained by the serial scaling results: dividing the number of rigid bodies per MPI process by two yields more than a factor of two in performance. The second observation is that the reduction of communication overhead by a smaller cut area improves the performance considerably.

Weak scaling in a rigid body simulation proves to be more complicated than strong scaling, because increasing the number of rigid bodies inevitably changes the simulated scenario due to additional interactions. It also raises the question of where to create the new bodies. Therefore we chose to use weak scaling to analyze the performance influence of the number of communication neighbors and to demonstrate simulations with several million rigid bodies.

In our weak scaling experiments, each process owns a cubic subdomain of initially $25 \times 25 \times 25$ spheres. These blocks of spheres are now put together depending on the arrangement of the processes. Note that, due to this setup, the weak scaling experiments cannot be directly compared to the strong scaling

Table 1. Strong scaling experiment of 1000 time steps with 65 536 spheres initially arranged in a 64x64x16 grid. The total volume of the spheres occupies approximately 1% of the domain volume.

# Cores	Partitioning	Runtime [s]
4	4 × 1	428.1
	2 × 2	351.0
8	8 × 1	201.1
	4 × 2	157.5
16	16 × 1	84.69
	8 × 2	62.38
	4 × 4	60.59
32	32 × 1	33.85
	16 × 2	24.95
	8 × 4	20.99

Table 2. Weak scaling experiment of 1000 time steps with up to 8 000 000 spheres. Each process initially owns a block of $25 \times 25 \times 25$ spheres. The total volume of the spheres occupies approximately 1% of the domain volume.

# Cores	# Spheres	Partitioning	Runtime [s]
64	1 000 000	64 × 1 × 1	273.2
		8 × 8 × 1	428.9
		4 × 4 × 4	609.7
128	2 000 000	128 × 1 × 1	278.5
		16 × 8 × 1	437.4
		8 × 4 × 4	641.1
256	4 000 000	256 × 1 × 1	279.9
		16 × 16 × 1	447.8
		8 × 8 × 4	645.5
512	8 000 000	512 × 1 × 1	278.5
		32 × 16 × 1	458.9
		16 × 8 × 4	670.6

experiments: whereas in both strong scaling experiments the shape of the domain was fixed and the shape of the subdomains was adapted according to the arrangement of the processes, in the weak scaling experiments the shape of a subdomain is fixed and the shape of the domain results from the arrangement of the processes.

The weak scaling experiments is shown in Table 2. The first observation is that the scaling behavior clearly depends on the number of communication neighbors: when choosing a particular number of processes, a one-dimensional partitioning is clearly the fastest, whereas a two-dimensional partitioning increases the maximum number of communication neighbors from two to eight. In case a three-dimensional partitioning is chosen, the maximum number of communication neighbors even increases to 26. Note that this results primarily from the choice of cubic blocks of spheres per process, which directly leads to an increased

communication overhead in case of more neighbors, since for every neighboring process, messages have to be individually encoded and decoded. The second observation is that different partitionings with a similar communication overhead result in approximately the same runtime. Considering the fact that every single measurement represents a different scenario, this is a result that demonstrates the generally good scaling behavior of the MPI parallelization of the *pe* rigid body engine.

The simulations with 512 cores deserve some special attention. Although these simulations do by far not show the limits of the *pe* framework, they demonstrate that it is possible to simulate several million interacting rigid bodies in reasonable runtimes.

5 Conclusions

In this paper we have presented a novel approach to large-scale rigid body simulations. With the algorithm presented, we are able to simulate several million rigid bodies on an arbitrary number of processes. This increase in the number of rigid bodies in a single simulation by at least two magnitudes in comparison to other algorithms enables for the first time the simulation of large-scale rigid body scenarios such as granular media, fluidization processes or sedimentation processes. The obvious advantage in comparison to other large-scale, particle-based simulation methods, for example molecular dynamics, the discrete element method, or smooth particle hydrodynamics, is that the geometry of the rigid bodies is fully resolved. Therefore it is possible to simulate these scenarios with arbitrarily shaped rigid bodies.

We have demonstrated the good scaling behavior of our algorithm. However, in this paper we have only used spheres, and we have not used any load balancing extension. Instead, we have focused on the analysis of the performance with fixed process boundaries. Further performance experiments should consider load balancing and investigate the influence of different geometries on the performance. Future work will include experiments on a larger number of cores and rigid body simulations of physically relevant scenarios.

References

1. Homepage of the Open Dynamics Engine (ODE), <http://www.ode.org/>
2. Homepage of the OpenTissue simulation framework, <http://www.opentissue.org>
3. Homepage of the Regional Computing Center Erlangen (RRZE),
<http://www.rrze.uni-erlangen.de>
4. Anitescu, M.: Optimization-based simulation of nonsmooth rigid multibody dynamics. *Math. Program.* 105(1), 113–143 (2006)
5. Cottle, R.W., Pang, J.S., Stone, R.E.: *The Linear Complementarity Problem*. Academic Press, Inc., London (1992)
6. Eberly, D.: *Game Physics*. Series in Interactive 3D Technology. Morgan Kaufmann, San Francisco (2003)

7. Erleben, K., Sporring, J., Henriksen, K.: Physics-Based Animation. Delmar (2005)
8. Fleissner, F., Eberhard, P.: Parallel load-balanced simulation for short-range interaction particle methods with hierarchical particle grouping based on orthogonal recursive bisection. *International Journal for Numerical Methods in Engineering* 74, 531–553 (2007)
9. Gropp, W., Skjellum, A., Lusk, E.: Using MPI: Portable Parallel Programming with the Message Passing Interface, 2nd edn. MIT Press, Cambridge (1999)
10. Iglberger, K., Thürey, N., Rüde, U.: Simulation of Moving Particles in 3D with the Lattice Boltzmann Method. *Computers & Mathematics with Applications* 55(7), 1461–1468 (2008)
11. Jean, M.: The non-smooth contact dynamics method. *Computer Methods in Applied Mechanics and Engineering* 177(3–4), 235–257 (1999)
12. Kaufman, D.M., Edmunds, T., Pai, D.K.: Fast frictional dynamics for rigid bodies. *ACM Transactions on Graphics (SIGGRAPH 2005)* 24, 946–956 (2005)
13. Millington, I.: Game Physics Engine Development. Series in Interactive 3D Technology. Morgan Kaufmann, San Francisco (2007)
14. Preclik, T.: Iterative rigid multibody dynamics. Diploma thesis, Friedrich-Alexander University of Erlangen-Nuremberg, Computer Science 10 – Systemsimulation (2008)
15. Renouf, M., Alart, P.: Conjugate gradient type algorithms for frictional multi-contact problems: applications to granular materials. *Computer Methods in Applied Mechanics Engineering* 194, 2019–2041 (2005)
16. Tasora, A., Negrut, D., Anitescu, M.: Large-scale parallel multi-body dynamics with frictional contact on the graphical processing unit. *Proceedings of the Institution of Mechanical Engineers, Part K: Journal of Multi-body Dynamics* 222(4), 315–326 (2008)
17. Wengenroth, H.: Rigid body collisions. Master's thesis, University of Erlangen-Nuremberg, Computer Science 10 – Systemsimulation 2007, Computer Science Department 10 (System Simulation), University of Erlangen-Nuernberg (2007)

Optimized Stencil Computation Using In-Place Calculation on Modern Multicore Systems

Werner Augustin¹, Vincent Heuveline², and Jan-Philipp Weiss¹

¹ SRG New Frontiers in High Performance Computing

² RG Numerical Simulation, Optimization, and High Performance Computing
Karlsruhe Institute of Technology, Universität Karlsruhe (TH), Germany
{werner.augustin,vincent.heuveline,jan-philipp.weiss}@kit.edu

Abstract. Numerical algorithms on parallel systems built upon modern multicore processors are facing two challenging obstacles that keep realistic applications from reaching the theoretically available compute performance. First, the parallelization on several system levels has to be exploited to the full extent. Second, provision of data to the compute cores needs to be adapted to the constraints of a hardware-controlled nested cache hierarchy with shared resources.

In this paper we analyze dedicated optimization techniques on modern multicore systems for stencil kernels on regular three-dimensional grids. We combine various methods like a compressed grid algorithm with finite shifts in each time step and loop skewing into an optimized parallel in-place stencil implementation of the three-dimensional Laplacian operator. In that context, memory requirements are reduced by a factor of approximately two while considerable performance gains are observed on modern Intel and AMD based multicore systems.

Keywords: Parallel algorithm, multicore processors, bandwidth-bound, in-place stencil, cache optimization, compressed grid, time blocking.

1 Introduction

The advent of multicore processors has brought the prospect of constantly growing compute capabilities at the price of ubiquitous parallelism. At least at a theoretical level, every new processor generation with shrinking die area and increasing core count is about to deliver the expected performance gain. However, it is well known that the main memory speed cannot keep up with the same rate of growth. Hence, the at all times existing memory gap becomes more and more pronounced. Hardware designers try to mitigate the gap by a hierarchy of nested and shared caches between the main memory and the highly capable compute cores. Moreover, sophisticated techniques and hardware protocols are developed in order to overcome existing bottlenecks. Hence, beside the issue of fine granular parallelism, the programmer has to consider the hierarchical and parallel memory design.

On modern multicore systems, on coprocessor devices like graphics processing units (GPUs), or on parallel clusters computation is overlapped with communication in order to optimize performance. However, many applications - especially numerical algorithms for the solution of partial differential equations - are bandwidth-bound in the sense that it takes more time to transfer data to the cores than it takes to process the same amount of data on the compute cores. The algorithms can be characterized by their computational intensity $I = f/w$ that is defined by the ratio of the number f of performed floating point operations (flop) per number w of transferred words (8 byte in double precision). For bandwidth-bound algorithms there is an upper bound for the effective performance P_{eff} given by the system bandwidth B (in byte/s) and the computational intensity I . This upper bound reads $P_{\text{eff}} \leq BI/8 = fB/(8w)$ for double precision and is often far less than the theoretically available compute performance. Basic routines of linear algebra like BLAS 1 or 2 routines have a computational intensity of order $O(1)$ with respect to the problem size N (i.e. vector size). Higher values can be for example achieved for lattice Boltzmann kernels (still $O(1)$) or FFT kernels ($O(\log N)$). For BLAS 3 routines, the computational intensity is asymptotically growing linearly in N . For particle methods computational intensity is also increasing with problem size. One of the worst case routines is the DAXPY operation with $I = 2/3$. For the seven-point Laplacian stencil considered in this work we find $I = 4$.

Due to the restricting properties of the algorithms under consideration, a hardware-aware and memory hierarchy-adapted parallel implementation is crucial for best possible performance. On local-store-based architectures like the STI Cell processor or stream computing platforms like GPUs all memory transfers are managed explicitly by the user or semi-implicitly by the software environment. On modern multicore processors data provision is handled by hardware-controlled caches where the cache structure is hidden and transparent to the programmer. Moreover, cache memory is not addressable.

The goal of this paper is an efficient implementation of a 3D stencil application on modern cache-based multicore systems where the OpenMP programming approach is chosen for parallelization. A considerable performance increase can be achieved by applying an in-place implementation with a single array on a compressed grid. As an additional benefit, memory usage is reduced by a factor of approximately two. Typical multicore optimization techniques are applied as well in that context. This work is extending part of the contribution of [3] where dedicated techniques are developed in a similar context assuming out-of-place implementation with Pthreads. In the comprehensive work [2] in-place implementations are addressed without giving details on the algorithm, its implementation or any performance results. Our contribution shows the clear advantage of this algorithmic modification. For more details on possible applications of the developed techniques as well as existing schemes we refer e.g. to [4, 6, 7, 8] and references therein.

This paper is organized as follows. Section 2 is dedicated to the framework of the considered stencil kernels. In Section 3 we provide a description of the

in-place stencil algorithm under consideration and describe the applied optimization techniques in more detail. Performance results are presented in Section 4 for different multicore systems ranging from Intel Clovertown and Nehalem, AMD Opteron, Barcelona and Shanghai, to Intel Itanium2. We conclude with a summary in Section 5.

2 Stencil Kernels

Stencil kernels are one of the most important routines applied in the context of structured grids [4] and arise in many scientific and technical disciplines. In image processing for example, stencils are applied for blurring and edge detection. More importantly, the discrete modeling of many physical processes in computational fluid dynamics (e.g. diffusion and heat transfer), mechanical engineering and physics (e.g. electromagnetism, wave theory, quantum mechanics) involves application of stencils. Stencils originate from the discretization of differential expressions in partial differential equations (PDEs) by means of finite element, finite volume, or finite difference methods. They are defined as fixed subset of nearest neighbors where the corresponding node values are used for computing weighted sums. The associated weights correspond to the coefficients of the PDEs that are assumed to be constant in our context. Stencil computations are endowed with a high spatial locality that expresses interactions restricted to nearest neighbors. Global coupling, as observed for the Laplacian/Poisson or the heat equation, is induced across several stencil iterations or by inversion of the associated matrix. In three dimensions 7-point (6+1), 9-point (8+1), or 27-point (6+8+12+1) stencils are common. The associated directions correspond to the faces, corners, and edges of a cube plus its center. Since the stencil only depends on the geometric location of the nodes, all necessary information can be deduced from the node indices. Due to the limited amount of operations per grid point with corresponding low computational intensity, stencil kernels are typically bandwidth-bound and have only a limited reuse of data. On most architectures stencil kernels are consequently running at a low fraction of theoretical peak performance.

In this paper, we consider a three-dimensional cubic grid with edge length N and a scalar value associated to each grid point. The grid is mapped to a one-dimensional array of size $n = N^3$ plus ghost layers. In our case we consider cyclic boundary conditions across the physical boundaries of the cubic domain. The 7-point Laplacian or heat equation stencil

$$U_{i,j,k}^{m+1} = \alpha U_{i,j,k}^m + \beta [U_{i+1,j,k}^m + U_{i-1,j,k}^m + U_{i,j+1,k}^m + U_{i,j-1,k}^m + U_{i,j,k+1}^m + U_{i,j,k-1}^m] \quad (1)$$

is accessing six neighbors with stride ± 1 , $\pm N$, and $\pm N^2$ in lexicographical order. These long strides require particular care to ensure data reuse since cache sizes are limited and much smaller than the problem size in general. In (1) $8n$ flop are performed on a vector of length n while at least $2n$ elements have to be transferred from main memory to the cores and back.

The considered Laplacian stencil scheme falls into the class of Jacobi iterations where all grid nodes can be updated independently and the order of the updates

does not matter. However, the updated nodes cannot be stored in the same array as the original values since some values are still needed to update other neighbors. For this reason so called out-of-place implementations with two different arrays for read and write operations are used. Parallelization of Jacobi-like methods by domain sub-structuring is straightforward since only ghost layers corresponding to the spatial and temporal width of the stencil need to be considered. For Gauss-Seidel-like stencil codes (e.g. for time-dependent problems) with interlaced time dependencies computation can be performed in-place with a single array for read and write operations, i.e. all updated values can be stored in the array of input values. However, the order of the grid traversal is decisive and the procedure is typically sequential. Multi-color approaches (red-black coloring) or wavefront techniques need to be considered for parallelization.

3 Description of the In-Place Stencil Kernel and Applied Optimization Techniques

Usually stencil operations are implemented as two-grid algorithms with two arrays where solutions of consecutive time steps are stored in alternating manner by swapping the arrays in each time step. As a consequence, data has to be read and written from and to different locations in main memory with significant delays due to compulsory cache misses caused by write operations. Considering the actual algorithm, these write cache misses are unnecessary. Therefore, attempts by using cache-bypassing mechanisms were made with varying success in [3] as well as in our study. Another possible remedy is to merge both arrays in an interleaved manner with some additional effort in the implementation [5]. But this procedure does not improve the additional memory requirements.

In order to tackle problems of memory consumption and pollution of memory transfers we are using an approach similar to the grid compression technique presented in [5] for lattice Boltzmann stencils. In this approach updated values are stored in-place and old input data is overwritten. Since grid point updates depend on all neighbors, a specified number of grid points has to be stored temporarily. This temporary storage within the same array results in a shift of the coordinates for the updated values. The underlying algorithm is detailed in the sequel.

3.1 One-Dimensional In-Place Computation

For a detailed description of the in-place stencil algorithm we start with the one-dimensional stencil operation

$$y[i] = \alpha x[i] + \beta(x[i - 1] + x[i + 1]). \quad (2)$$

Simple replacement of $y[i]$ by $x[i]$ yields unresolvable dependencies. But if the array is skewed with a shift of one there are no more dependencies considering

$$x[i - 1] = \alpha x[i] + \beta(x[i - 1] + x[i + 1]). \quad (3)$$

a₀	a₁	a ₂	a ₃	a ₄	a ₅	a ₆	a ₇	a ₈	a ₉	a ₁₀	a ₁₁		
a₀	a₁	a₂	a₃	a₄	a₅	a ₆	a ₇	a ₈	a ₉	a ₁₀	a ₁₁	a₀	a₁
b₁	b₂	b₃	b₄	b₅	b₆	b₇	b₈	a₈	a₉	a₁₀	a₁₁	a ₀	a ₁
b ₁	b ₂	b ₃	b ₄	b₅	b₆	b₇	b₈	a₈	a₉	a₁₀	a₁₁	a₀	a₁
b ₁	b ₂	b ₃	b ₄	b ₅	b ₆	b ₇	b ₈	b₉	b₁₀	b₁₁	b₀	a ₀	a ₁
b ₁	b ₂	b ₃	b ₄	b ₅	b ₆	b ₇	b ₈	b ₉	b ₁₀	b ₁₁	b ₀		

Fig. 1. One-dimensional, space-blocked, and shifted stencil operation

a₀	a₁	a₂	a₃	a₄	a₅	a ₆	a ₇	a ₈	a ₉	a ₁₀	a ₁₁		
a₀	a₁	a₂	a₃	a₄	a₅	a ₆	a ₇	a ₈	a ₉	a ₁₀	a ₁₁	a₀	a₁
b₁	b₂	b₃	b₄	a ₄	a ₅	a ₆	a ₇	a ₈	a ₉	a ₁₀	a ₁₁	a ₀	a ₁
c₂	c₃	b₃	b₄	a₄	a₅	a₆	a₇	a₈	a₉	a ₁₀	a ₁₁	a ₀	a ₁
c ₂	c ₃	b₃	b₄	b₅	b₆	b₇	b₈	a ₈	a ₉	a ₁₀	a ₁₁	a ₀	a ₁
c₂	c₃	c₄	c₅	c₆	c₇	b₇	b₈	a ₈	a ₉	a ₁₀	a ₁₁	a ₀	a ₁
d₃	d₄	d₅	d₆	c ₆	c ₇	b₇	b₈	a₈	a₉	a₁₀	a₁₁	a₀	a₁
d ₃	d ₄	d ₅	d ₆	c ₆	c ₇	b₇	b₈	b₉	b₁₀	b₁₁	b₀	a ₀	a ₁
d ₃	d ₄	d ₅	d ₆	c₆	c₇	c₈	c₉	c₁₀	c₁₁	b ₁₁	b ₀	a ₀	a ₁
d ₃	d ₄	d ₅	d ₆	d₇	d₈	d₉	d₁₀	c ₁₀	c ₁₁	b₁₁	b ₀	a₀	a₁
d ₃	d ₄	d ₅	d ₆	d ₇	d ₈	d ₉	d ₁₀	c₁₀	c₁₁	c₀	c₁	c₂	c₃
d ₃	d ₄	d ₅	d ₆	d ₇	d ₈	d ₉	d ₁₀	d₁₁	d₀	d₁	d₂	c ₂	c ₃
d ₃	d ₄	d ₅	d ₆	d ₇	d ₈	d ₉	d ₁₀	d ₁₁	d ₀	d ₁	d ₂		

Fig. 2. One-dimensional, space-blocked, time-blocked and shifted stencil operation

With additional spatial blocking included the computation scheme is shown in Figure 11. The different rows represent snap-shots of the grid array after a blocked computation. Lower indices denote the grid coordinate in the problem domain. Letters *a*, *b*, *c*, etc., indicate the values of the original and subsequent time steps. The current changes are highlighted with dark gray background, the cells providing input for the next computation are highlighted with lighter gray. Additionally, cells at the block boundaries touched only once are marked with lightest gray. The first step consists of copying the left boundary to the right to allow for computation of cyclic boundary conditions. The consecutive steps show blocked and shifted computations. In the last row all updated values of the next time steps are available. The whole domain is shifted to the left by one.

In the next example in Figure 2 time-blocking is added. Here, three consecutive time steps are computed immediately within one block in order to improve data reuse and cache usage. Every block update still follows formula (3), i.e. every update uses data from itself and the two neighboring cells to the right,

a₀	a₁	a₂	a₃	<i>a₄</i>	<i>a₅</i>					a₆	a₇	a₈	a₉	<i>a₁₀</i>	<i>a₁₁</i>				
a₀	a₁	a₂	a₃	a₄	a₅	a₆	a₇	a₈	a₉	a₆	a₇	a₈	a₉	a₁₀	a₁₁	a₀	a₁	a₂	a₃
b₁	b₂	b₃	b₄	<i>a₄</i>	<i>a₅</i>	<i>a₆</i>	<i>a₇</i>	<i>a₈</i>	<i>a₉</i>	b₇	b₈	b₉	b₁₀	<i>a₁₀</i>	<i>a₁₁</i>	<i>a₀</i>	<i>a₁</i>	<i>a₂</i>	<i>a₃</i>
c₂	c₃	<i>b₃</i>	<i>b₄</i>	a₄	a₅	a₆	a₇	a₈	a₉	c₈	c₉	<i>b₉</i>	<i>b₁₀</i>	a₁₀	a₁₁	a₀	a₁	a₂	a₃
<i>c₂</i>	<i>c₃</i>	b₃	b₄	b₅	b₆	b₇	b₈	<i>a₈</i>	<i>a₉</i>	<i>c₈</i>	<i>c₉</i>	b₉	b₁₀	b₁₁	b₀	b₁	b₂	<i>a₂</i>	<i>a₃</i>
<i>c₂</i>	<i>c₃</i>	c₄	c₅	c₆	c₇					<i>c₈</i>	<i>c₉</i>	c₁₀	c₁₁	c₀	c₁	<i>b₁</i>	<i>b₂</i>	<i>a₂</i>	<i>a₃</i>
<i>c₂</i>	<i>c₃</i>	<i>c₄</i>	<i>c₅</i>	<i>c₆</i>	<i>c₇</i>					<i>c₈</i>	<i>c₉</i>	<i>c₁₀</i>	<i>c₁₁</i>	<i>c₀</i>	<i>c₁</i>				

Fig. 3. One-dimensional, space-blocked, time-blocked, and shifted stencil operation on two cores

resulting in a skew of two between consecutive block computations (skew is one for out-of-place time skewing with two grids). Data flows from light gray to dark gray again. Light gray cells can be interpreted as cache misses because they are used but have not been computed in the previous block computation. As can be seen from the figure, cache misses are only slightly increased in case of temporal blocking. The final result is shifted to the left by the number of blocked time steps (by three in our case).

Our third example in Figure 3 shows the boundary treatment at the domain interfaces between different cores. The boundary overlap of the previous example has to be replicated for every core to allow parallel computation. This results in the fact that a number of boundary cells (equal to the number of blocked time steps) have to be calculated on both involved cores. This leads to some small computational overhead.

3.2 Multi-dimensional In-Place Computation

The results of the previous section can be applied to the two-dimensional case in a straightforward way. The corresponding computational pattern is shown in Figure 4. Different shades of gray denote the sequence of computation for the respective blocks.

The three-dimensional case is exemplified by means of a C code snippet in Figure 5 where initialization of appropriate boundaries is omitted. Parameters

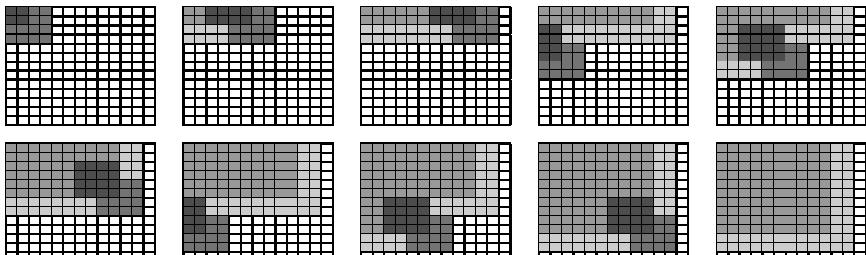


Fig. 4. Two-dimensional, space-blocked, time-blocked and shifted stencil operation

```

for( bz = 0; bz < zsize ; bz += bsz )
  for( by = 0; by < ysize; by += bsy )
    for( bx = 0; bx < xsize; bx += bsx )
      for( t = 0; t < tmax; t++ ) {
        tc = tmax - t - 1;
        xmin = max(0, bx - 2*t); xmax = min(xsize + 2*tc, bx+bsx - 2*t);
        ymin = max(0, by - 2*t); ymax = min(ysize + 2*tc, by+bsy - 2*t);
        zmin = max(0, bz - 2*t); zmax = min(zsize + 2*tc, bz+bsz - 2*t);
        for( z = zmin; z < zmax; z++ )
          for( y = ymin; y < ymax; y++ )
            for( x = xmin; x < xmax; x++ )
              a[x, y, z] = stencil(a[x+1, y+1, z+1],
                                    a[x , y+1, z+1], a[x+2, y+1, z+1], a[x+1, y , z+1],
                                    a[x+1, y+2, z+1], a[x+1, y+1, z ] , a[x+1, y+1, z+1]);
      }
    }
  }
}

```

Fig. 5. C example code for the three-dimensional, space-blocked, time-blocked and shifted stencil operation

`xsize`, `ysize` and `zsize` denote the size of the computational domain. The size of the spatial block is specified by `bsx`, `bsy` and `bsz`. And finally, `tmax` is the number of blocked time steps.

3.3 Additional Optimization Techniques

In the following list applied multicore optimization techniques are given:

- **Parallel data allocation** is essential for performance on NUMA-architectures. Therefore, threads are pinned to specific cores and memory initialization is explicitly done by the same mapping.
- **Time and space blocking** is applied for reuse of (2nd level) cached data.
- **Cache-bypassing** is used to prevent unnecessary cache line loads due to write misses. Current processors access main memory in the granularity of cache lines (usually 128 bytes). Writing one double floating point value (8 bytes) therefore involves reading a complete cache line from memory, changing the corresponding bytes and writing the whole line back. Special SSE-instructions (in our context `mm_stream_pd`) allow to bypass the cache and write to memory directly, without wasting memory bandwidth for data which will be instantly overwritten anyway.
- **Parallelization** is done only along the z -axis. On a 512^3 grid with five blocked time steps an overhead of approximately 16% due to redundant computation at overlapping domains between processor cores is introduced (in the case of 8 threads). Parallelization along all axes would reduce this overhead to approximately 12%. With only three blocked time steps both values decrease to 9% and 7%.

Optimizations not considered in our implementation are listed below:

- Based on the results in [3] **SIMDization** was not considered to be valuable enough.
- **Register blocking** was briefly checked but is not used because no performance gain was observed (actually there is an implicit register blocking of two in x -direction when using the SSE-instruction for cache-bypassing).
- **Explicit software prefetching** is not used although a comparison with upper performance bounds in Section 4 suggests to investigate this topic.

4 Performance Results

In this section we present performance results on the platforms described in Section 4.1. Our goal is to evaluate and validate the proposed approach across different platforms. For all test runs we consider stencil computation on a 512^3 grid which requires at least 1 GB of memory.

The left diagram of each figure compares an out-of-place implementation (with and without cache bypassing) and our in-place algorithm in terms of grid updates. Three relevant vector operations (copy operation $y[i] = x[i]$ with and without cache bypassing and vector update $x[i] = x[i] + a$) are added for reference. These results should be interpreted as upper bounds representing the minimum necessary memory operations. Grid updates are chosen as unit of comparison in order to avoid confusion between single- and bi-directional memory transfer rates. The right diagram compares an out-of-place implementation with and without cache bypassing and our in-place algorithm in terms of GFlop/s. On top of the single time step results performance improvements for different time-blocking parameters are shown. A single grid node update counts 8 flop.

4.1 Test Environment: Hardware and Software

Our approach of an in-place stencil algorithm was tested and compared to other algorithms on a couple of cache-based multicore architectures:

- **2-way Intel Clovertown:** a 2-way SMP node with quad-core Intel Xeon X5355 processors running at 2.66 GHz clock frequency
- **2-way Intel Nehalem:** a 2-way SMP node with quad-core Intel Xeon E5520 processors running at 2.27 GHz clock frequency
- **2-way AMD dual-core Opteron:** an HP ProLiant DL145 G2 server, a 2-way SMP node with AMD dual-core Opteron 285 processors with 2.66 GHz clock frequency
- **4-way AMD dual-core Opteron:** an HP ProLiant DL585 server, a 4-way SMP node with AMD dual-core Opteron 8218 processors with 2.66 GHz clock frequency
- **4-way AMD quad-core Barcelona:** a 4-way SMP node with AMD quad-core Opteron 8350 (Barcelona) processors running at 2.0 GHz clock frequency

- **2-way AMD quad-core Shanghai:** a 2-way SMP node with AMD quad-core Opteron 2384 (Shanghai) processors running at 2.7 GHz clock frequency
- **8-way single core Intel Itanium2:** an HP Integrity RX8620 server, a 16-way single core Itanium2 Madison server partitioned in two 8-way nodes with 1.6 GHz clock frequency.

Beside the Intel Clovertown and Itanium2 (8 processors connected to two memory banks in interleave mode) all SMP systems are typical NUMA architectures with one memory bank per processor. All stencil algorithms are implemented in *C* using *OpenMP* as parallelization paradigm. They are compiled using an Intel C compiler version 11.0. On the platforms with Intel processors the optimization flag `-fast` is used. The code for the AMD processor based platforms is compiled with `-O3 -ipo -static -no-prec-div -xW`. On the Itanium2 platform we revert to the Intel C compiler version 10.1 because of severe performance problems. Presumably the compiler fails to get the software prefetching right resulting in 30-50% less performance.

4.2 Stencil Performance Across Several Platforms

The Intel Clovertown results are shown in Figure 6. The saturation point of the memory bandwidth is almost reached with two cores. For four and eight cores the conventional out-of-place results agree very well with the ones presented in [3] and draw near the upper bound. As also mentioned in the article referenced above, adding cache-bypassing shows the same disappointingly low performance improvements. In-place computation improves these results by about 30%. The simpler memory access pattern also improves the effect of time-blocking considerably reaching an improvement of approximately 70%.

The results for the Intel Nehalem in Figure 7 show a performance boost of a factor of three compared to the Clovertown results. Here, the integrated memory controller brings considerable improvements for this bandwidth-bound kernel. The last column in each plot shows minor benefits or even a degradation from *Hyper-Threading* (HT) with two threads per core. There is no considerable profit

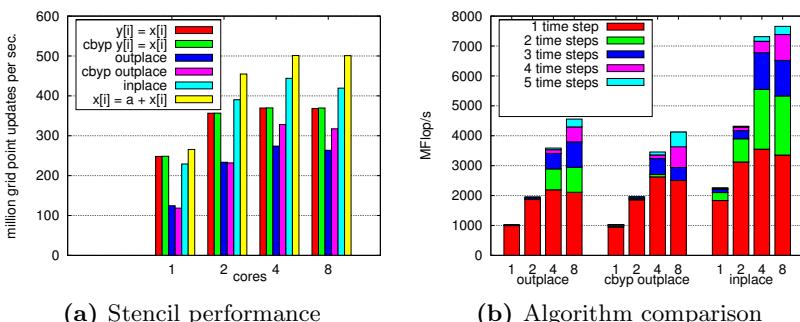


Fig. 6. Intel Clovertown, quad-core, 2-way node, 512x512x512 grid

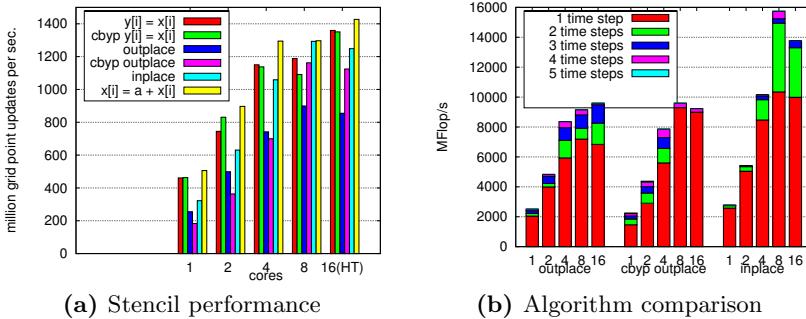


Fig. 7. Intel Nehalem, quad-core, 2-way node, 512x512x512 grid

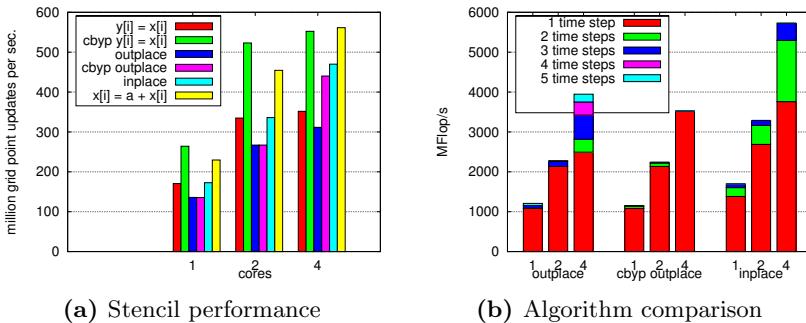


Fig. 8. AMD Opteron, dual-core, 2-way node, 512x512x512 grid

from time-blocking in the out-of-place version of the algorithm. Cache-bypassing becomes effective when all cores are running. We find significant performance improvements when ramping up from two to eight cores.

The results of the AMD dual-core processors are shown in Figure 8 and Figure 9. Results for the 4-way node are considerably better for the case of 4 cores because of the total of 4 memory banks instead of 2 for the 2-way node. The performance improvement between the out-of-place and in-place variants are at relatively low 7% for the version without time-blocking. This benefit increases to 40% for the 2-way nodes (13% and 27% for the 4-way nodes respectively) in case of time blocking.

The results of the AMD quad-core Barcelona processors are shown in Figure 10. While the results for the conventional out-of-place algorithm are similar to the ones presented in [3] (though one should be careful because of the hardware differences and the different clock speeds) the additional cache-bypassing shows very little improvement in our study. Possibly, non-optimal SSE instructions were used. The in-place results also lag behind the potential upper bound marked by the respective vector operations. Like on all the other processors

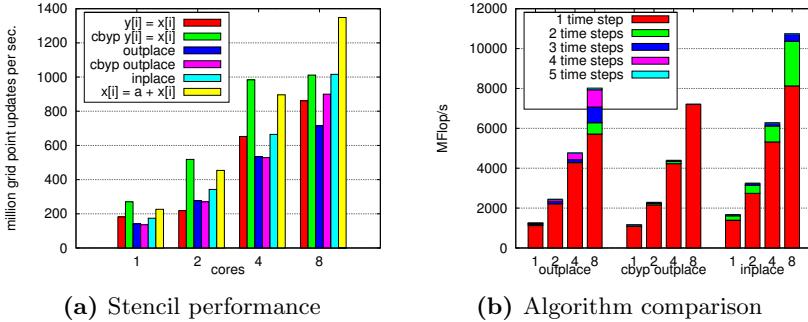


Fig. 9. AMD Opteron, dual-core, 4-way node, 512x512x512 grid

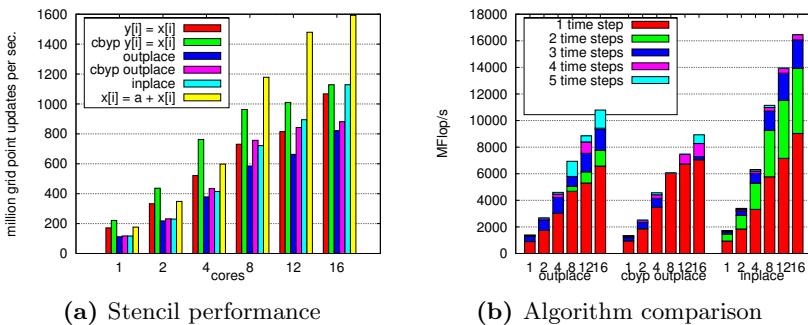


Fig. 10. AMD Barcelona, quad-core, 4-way node, 512x512x512 grid

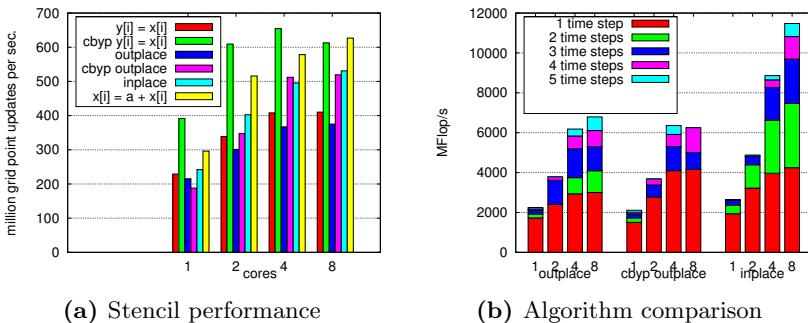


Fig. 11. AMD Shanghai, quad-core, 2-way node, 512x512x512 grid

analyzed so far, time-blocking profits much stronger from the in-place algorithm than from the out-of-place version.

The results of the AMD Shanghai in Figure 11 clearly lag behind those of its competitor Intel Nehalem. Compared to the AMD Barcelona results we find a degradation by a factor of approximately 1.5 that is mainly attributed to the

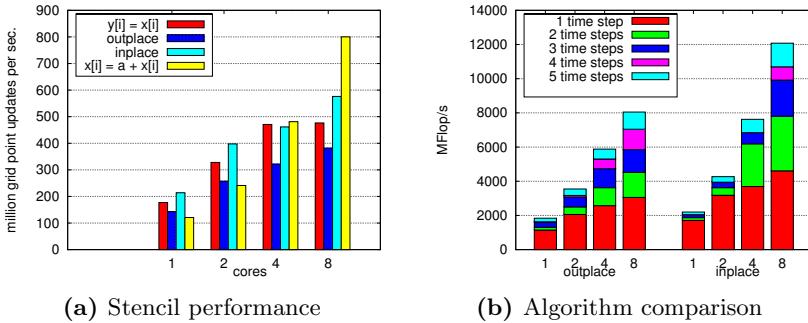


Fig. 12. Intel Itanium2, single-core, 8-way node, 512x512x512 grid

2-way and 4-way memory access. Performance values obtained by using cache-bypassing are very close to those from the in-place version.

Finally, the results of the Itanium2 architecture are presented in Figure 12. Due to the EPIC-architecture of the Itanium processor (EPIC = explicitly parallel instruction computing) performance heavily depends on compiler optimization which often conflicts with heuristics of memory and register resource allocation introduced by OpenMP. The compiler can be caused to disable essential techniques like software prefetching. This is a possible explanation why even the simple vector operations used for reference show bad performance. This observation is underlined by huge backdrops in performance for different versions of the Intel C compiler.

5 Conclusion

In this work we analyze the performance of an in-place implementation of a 3D stencil kernel on cache-based multicore systems using OpenMP. The proposed approach leads to considerable performance improvements especially considering combined time and space blocking techniques. In a future work the proposed scheme will serve as a building block for a larger Navier-Stokes solver on structured grids. Furthermore, an extension for Dirichlet and Neumann boundary conditions as opposed to the cyclic boundary conditions considered in this paper is currently under development.

Acknowledgements

The Shared Research Group 16-1 received financial support by the Concept for the Future of Karlsruhe Institute of Technology in the framework of the German Excellence Initiative and the industrial collaboration partner Hewlett-Packard.

We also thank the Steinbuch Centre for Computing (SCC) at Karlsruhe Institute of Technology (KIT) for its support and providing the computing nodes of

its clusters. The server with the AMD Barcelona cores was made available by the Lehrstuhl Informatik für Ingenieure und Naturwissenschaftler of the Department of Computer Science at KIT.

References

1. Asanovic, K., Bodik, R., Catanzaro, B., Gebis, J., Husbands, P., Keutzer, K., Patterson, D., Plishker, W., Shalf, J., Williams, S., Yelick, K.: The landscape of parallel computing research: a view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley (December 2006)
2. Datta, K., Kamil, S., Williams, S., Oliker, L., Shalf, J., Yelick, K.: Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Review* 51(1), 129–159 (2009)
3. Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., Yelick, K.: Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In: *SC 2008: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pp. 1–12. IEEE Press, Los Alamitos (2008)
4. Kamil, S., Datta, K., Williams, S., Oliker, L., Shalf, J., Yelick, K.: Implicit and explicit optimizations for stencil computations. In: *MSPC 2006: Proceedings of the 2006 workshop on memory system performance and correctness*, pp. 51–60. ACM Press, New York (2006)
5. Th. Pohl, M., Kowarschik, J., Wilke, K.: Iglberger, and U. Rüde. Optimization and profiling of the cache performance of parallel lattice Boltzmann codes. *Parallel Processing Letters* 13(4) (December 2003)
6. Vuduc, R., Demmel, J., Yelick, K.: OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series* 16, 521–530 (2005)
7. Whaley, R., Petitet, A., Dongarra, J.: Automated empirical optimizations of software and the ATLAS project. *Parallel Computing* 27 (2001)
8. Wonnacott, D.: Time skewing for parallel computers. In: *Proc. 12th Workshop on Languages and Compilers for Parallel Computing*, pp. 477–480. Springer, Heidelberg (1999)

Parallel Implementation of Runge–Kutta Integrators with Low Storage Requirements

Matthias Korch and Thomas Rauber

University of Bayreuth, Department of Computer Science
`{korch,rauber}@uni-bayreuth.de`

Abstract. This paper considers the parallel solution of large systems of ordinary differential equations (ODEs) which possess a special access pattern by explicit Runge–Kutta (RK) methods. Such systems may arise, for example, from the semi-discretization of partial differential equations (PDEs). We propose an implementation strategy based on a pipelined processing of the stages of the RK method that does not impose restrictions on the choice of coefficients of the RK method. This approach can be implemented with low storage while still allowing efficient step control by embedded solutions.

1 Introduction

We consider the parallel solution of initial value problems (IVPs) of ordinary differential equations defined by

$$\mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}(t)) , \quad \mathbf{y}(t_0) = \mathbf{y}_0 , \quad \mathbf{y} : \mathbb{R} \rightarrow \mathbb{R}^n , \quad \mathbf{f} : \mathbb{R} \times \mathbb{R}^n \rightarrow \mathbb{R}^n . \quad (1)$$

Parallel numerical solution methods for such problems have been addressed by many authors. An overview is presented in [1]. In this paper, we focus on low-storage implementations of RK methods suitable for large systems of ODEs where storage space may be the determining factor in the choice of method and implementation variant.

Classical explicit s -stage RK methods with method coefficients a_{li} , c_i , and b_l compute, at each stage $l = 1, \dots, s$, starting with the approximation value $\boldsymbol{\eta}_\kappa \approx \mathbf{y}(t_\kappa)$, an argument vector

$$\mathbf{w}_l = \boldsymbol{\eta}_\kappa + h_\kappa \sum_{i=1}^{l-1} a_{li} \mathbf{v}_i \quad (2)$$

that is used as input for the function evaluation \mathbf{f} to compute a stage vector

$$\mathbf{v}_l = \mathbf{f}(t_\kappa + c_l h_\kappa, \mathbf{w}_l) . \quad (3)$$

Finally, the new approximation $\boldsymbol{\eta}_{\kappa+1} \approx \mathbf{y}(t_{\kappa+1})$ is computed by:

$$\boldsymbol{\eta}_{\kappa+1} = \boldsymbol{\eta}_\kappa + h_\kappa \sum_{l=1}^s b_l \mathbf{v}_l . \quad (4)$$

If the RK method provides an embedded solution, a second approximation $\hat{\eta}_{\kappa+1}$ can be computed using different weights \hat{b}_l . This second approximation allows an inexpensive estimation of the local error and, hence, an efficient stepsize control.

In the general case, where no assumptions about the method coefficients or the access pattern of the function evaluation can be made, an implementation of the classical RK scheme needs to hold at least $s + 1$ vectors of size n (so called *registers*) to store η_κ , $\mathbf{w}_2, \dots, \mathbf{w}_s$, and $\eta_{\kappa+1}$, where n is the dimension of the ODE system. One additional register ($\hat{\eta}_{\kappa+1}$ or an error estimate) is required for the implementation of a stepsize controller which can reject and repeat steps.

On modern computer systems the classical approach can be applied to ODE systems with tens or even hundreds of millions of components, since these systems are equipped with several GB of memory, even if a method of high order with a large number of stages is required. Some ODE problems, however, require an even larger number of components. These are, in particular, ODE systems derived from PDE systems by a spatial discretization using the method of lines. For example, a spatial resolution of 1 % leads to a system with $n_v \cdot 10^6$ components for a 2-dimensional PDE system and $n_v \cdot 10^9$ components for a 3-dimensional PDE system, where n_v is the number of dependent variables in the PDE system. Increasing the spatial resolution by a factor of 10 leads to an increase in the number of ODE components by a factor of 10^2 for the 2-dimensional PDE system and by a factor of 10^3 for the 3-dimensional PDE system.

In the following, after a discussion of related approaches to reduce the storage space of RK methods in Section 2, we present a new approach based on a pipelined processing of the stages in Section 3. This pipelining approach is applicable to ODE systems with a special access pattern as it is typical for ODE systems derived by the method of lines. It reduces the storage space to a single register plus $\Theta(\frac{1}{2}s^2 \cdot d(\mathbf{f}))$, where $d(\mathbf{f})$ is the access distance of the right-hand-side function \mathbf{f} (see Section 3.1), while preserving all degrees of freedom in the choice of RK coefficients. Stepsize control based on an embedded solution can be realized with only one additional register. Further, a parallel implementation of the pipelining scheme is possible (Section 3.5), which can efficiently utilize parallel computing resources and may be used to reduce the memory requirements of a single computing node even further. Section 4 compares the amount of memory required by the low-storage pipelining implementation with the amount required by other low-storage RK algorithms and by conventional implementations. An experimental evaluation of runtime and scalability of the new implementation is presented in Section 5. Finally, Section 6 concludes the paper.

2 Related Work

A first approach to get along with low storage space is the use of RK methods with a small number of stages. This, however, means abandoning desirable properties of the method, such as a high order. Therefore, several authors, e.g., [23456], propose special RK methods that can be implemented with low storage space (e.g., 2 or 3 registers) even though they have a larger number of stages

and a higher order. This is possible by the construction of special coefficient sets such that, at each stage, only the data stored in the available registers is accessed, and the values computed at this stage fit in the available registers. These approaches, however, retain some deficiencies:

1. The coefficient sets must conform to additional constraints. Hence, fewer degrees of freedom are available in the construction of the coefficients. As the result, the methods proposed have weaker numerical properties than classical RK schemes with the same number of stages. If stronger numerical properties are required, one must resort to classical RK schemes with higher memory usage.
2. Using a higher number of stages to reach a specific order leads to higher computational costs and, as a consequence, to a higher runtime.
3. Many of the low-storage RK methods proposed do not provide embedded solutions. Therefore, these methods are suitable only for fixed-stepsize integration, unless additional computations and storage space are invested into stepsize control (e.g., using Richardson extrapolation).
4. Low-storage RK methods with embedded solutions impose additional constraints on the method coefficients [7], thus further reducing the degrees of freedom in the construction of the coefficients.
5. Stepsize control requires additional registers: One additional register is needed to estimate the local error ($\hat{\eta}_{\kappa+1}$ or, alternatively, $\mathbf{e} = \hat{\eta}_{\kappa+1} - \eta_{\kappa+1}$). This register may be omitted if the embedded solution $\hat{\eta}_{\kappa+1}$ is computed at stage $s - 1$, so that it can be compared with $\eta_{\kappa+1}$, which is computed at stage s , without explicitly storing it [7]. In order to be able to repeat the time step, a second additional register is needed that saves η_{κ} .
6. Some of the 2-register low-storage RK schemes proposed rely on a specific structure of the coupling between the equations in the ODE system such that, for example, the argument vector of a function evaluation can be overwritten by the result of the function evaluation. If right-hand-side functions with arbitrary access patterns are to be supported, an additional register must be used to store the result of the function evaluation temporarily [2].

It is, therefore, desirable to find new methods or algorithms which overcome these deficiencies. In this paper, we reconsider the pipelining approach suggested, initially, to improve locality and scalability of embedded RK implementations in [8,9] from this perspective.

3 Block-Based Pipelining Approach with Reduced Storage Space

Many sparse ODEs, in particular many discretized PDEs derived by the method of lines, are described by a right-hand-side function $\mathbf{f} = (f_1, \dots, f_n)$ where the components of the argument vector accessed by each component function f_j lie within a bounded index range near j . In the following we review the pipelining approach proposed in [8,9], which exploits this property of the ODE system and

supports arbitrary RK coefficients. We further show how the pipelining approach can be implemented with stepsize control using only ‘2+’ registers, i.e., 2 registers plus some small extra space that vanishes asymptotically if the ODE system is very large. Without stepsize control the storage space required even reduces to ‘1+’ registers.

3.1 Access Distance of a Right-Hand-Side Function

While, in general, the evaluation of one component of the right hand-side-function \mathbf{f} , $f_j(t, \mathbf{w})$, may access all components of the argument vector \mathbf{w} , many ODE problems only require the use of a limited number of components. In many cases, the components accessed are located nearby the index j . To measure this property of a function \mathbf{f} , we use the *access distance* $d(\mathbf{f})$ which is the smallest value b , such that all component functions $f_j(t, \mathbf{w})$ access only the subset $\{w_{j-b}, \dots, w_{j+b}\}$ of the components of their argument vector \mathbf{w} . We say the access distance of \mathbf{f} is *limited* if $d(\mathbf{f}) \ll n$.

3.2 Pipelining Computation Order

The pipelining computation order described in the following can be applied if the right-hand-side function has a limited access distance. We subdivide all n -vectors into $n_B = \lceil n/B \rceil$ blocks of size B , where $d(\mathbf{f}) \leq B \ll n$. Then, the function evaluation of a block $J \in \{1, \dots, n_B\}$ defined by

$$\begin{aligned} \mathbf{f}_J(t, \mathbf{w}) = & (f_{(J-1)B+1}(t, \mathbf{w}), f_{(J-1)B+2}(t, \mathbf{w}), \dots, \\ & f_{(J-1)B+\min\{B, n-(J-1)B\}}(t, \mathbf{w})) \end{aligned}$$

uses only components of the blocks $J-1$, J , and $J+1$ of \mathbf{w} if these blocks exist, i.e., if $1 < J < n_B$. This dependence structure enables the computation of the blocks of the argument vectors $\mathbf{w}_2, \dots, \mathbf{w}_s$, the vector $\Delta\boldsymbol{\eta} = \boldsymbol{\eta}_{\kappa+1} - \boldsymbol{\eta}_\kappa$ and the error estimate $\mathbf{e} = \hat{\boldsymbol{\eta}}_{\kappa+1} - \boldsymbol{\eta}_{\kappa+1}$ as illustrated in Fig. 1(a). The computation starts with the first and the second block of \mathbf{w}_2 , which only requires components of $\mathbf{w}_1 = \boldsymbol{\eta}_\kappa$. Then, the first block of \mathbf{w}_3 can be computed since it only uses components of the first two blocks of \mathbf{w}_2 . In the next step, the third block of \mathbf{w}_2 is computed which enables the computation of the second block of \mathbf{w}_3 which again enables the computation of the first block of \mathbf{w}_4 . This is continued until the computation of the first two blocks of \mathbf{w}_s has been completed and the first block of $\Delta\boldsymbol{\eta}$ and \mathbf{e} has been computed. Then the next block of $\Delta\boldsymbol{\eta}$ and \mathbf{e} can be determined by computing only one additional block of $\mathbf{w}_2, \dots, \mathbf{w}_s$. This computation is repeated until the last block of $\Delta\boldsymbol{\eta}$ and \mathbf{e} has been computed.

3.3 Working Space of the Pipelining Scheme

An important advantage of the pipelining approach is that only those blocks of the argument vectors are kept in the cache which are needed for further computations of the current step. One step of the pipelining computation scheme

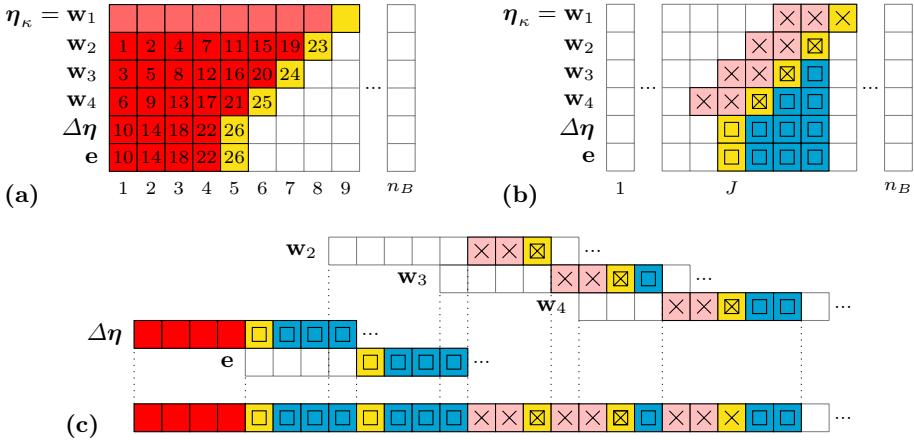


Fig. 1. (a) Illustration of the computation order of the pipelining computation scheme. After the blocks with index 4 of $\Delta\eta$ and \mathbf{e} have been computed, the next pipelining step computes the blocks with index 5 of $\Delta\eta$ and \mathbf{e} . (b) Illustration of the working space of one pipelining step. Blocks required for function evaluations are marked by crosses. Blocks updated using results of function evaluations are marked by squares. (c) Illustration of the overlapping of the vectors $\Delta\eta$, \mathbf{e} and $\mathbf{w}_2, \dots, \mathbf{w}_s$ in the low-storage implementation.

computes s argument blocks and one block of $\Delta\eta$ and \mathbf{e} . Since the function evaluation of one block J accesses the blocks $J-1$, J , and $J+1$ of the corresponding argument vector, $3s$ argument vector blocks must be accessed to compute one block of $\Delta\eta$ and \mathbf{e} . Additionally,

$$\sum_{i=3}^s (i-2) = \frac{1}{2}(s-1)(s-2) = \frac{1}{2}s^2 - \frac{3}{2}s + 1$$

blocks of argument vectors, s blocks of $\Delta\eta$ and s blocks of \mathbf{e} must be updated. Altogether, the working space of one pipelining step consists of

$$\frac{1}{2}s^2 + \frac{7}{2}s + 1 = \Theta\left(\frac{1}{2}s^2\right) \quad (5)$$

blocks of size B , see Fig. 1(b). Since $B \ll n$, this is usually only a small part of the working space of $\Theta(sn)$ adherent to general implementations suitable for arbitrary right-hand-side functions which iterate over all s argument vectors in the outermost loop.

3.4 Low-Storage Implementation of the Pipelining Scheme

Reconsidering the pipelining computation scheme in terms of storage space, we observe that the pipelining computation scheme, by performing a diagonal sweep across the argument vectors, delivers the blocks of elements of $\Delta\eta$ and \mathbf{e} one after

the other as the sweep proceeds along the system dimension. The computation of the next blocks of these vectors depends only on those argument vector blocks that belong to the working space of the corresponding pipelining step. Argument vector blocks behind the sweep line that is determined by the working space of the current pipelining step are no longer required for computations in the current time step. Argument vector blocks ahead of the sweep line do not yet contain data computed in the current time step and do not contribute to the computations of the current pipelining step. Hence, a significant reduction of the storage space could be achieved if only those argument vector blocks were kept in memory which are part of the working space of the current pipelining step.

A second opportunity to save storage space offers itself in the computation of the error estimate \mathbf{e} . The error estimate \mathbf{e} is used in the decision whether the current step size is to be accepted or whether it is to be rejected and should be repeated with a smaller stepsize. Most step control algorithms make this decision based on a scalar value that is an aggregate of the elements of \mathbf{e} , e.g., a vector norm. A common choice for the aggregate (ϵ) is the maximum norm of \mathbf{e}/\mathbf{s} divided by some user defined tolerance TOL :

$$\epsilon = \frac{1}{\text{TOL}} \left\| \frac{\mathbf{e}}{\mathbf{s}} \right\|_{\infty} = \frac{1}{\text{TOL}} \max_{j=1,\dots,n} \left| \frac{e_j}{s_j} \right| ,$$

where \mathbf{s} is a vector of scaling factors, e.g., $s_j = \max\{|\eta_{\kappa,j}|, |\eta_{\kappa+1,j}|\}$ for $j = 1, \dots, n$. But the computation of aggregates of this kind does not require all elements of \mathbf{e} to be stored in memory. Rather, a sweep approach can be applied which updates a scalar variable as the elements of \mathbf{e} are computed one after the other.¹

$$1.) \ \epsilon := 0 , \quad 2.) \ \epsilon := \max \left\{ \epsilon, \left| \frac{e_j}{s_j} \right| \right\} \text{ for } j := 1, \dots, n , \quad 3.) \ \epsilon := \frac{\epsilon}{\text{TOL}} .$$

This sweep approach to compute ϵ can easily be integrated into the pipelining scheme, which delivers one new block of \mathbf{e} after each pipelining step.

All in all, it is sufficient to store the two n -vectors $\boldsymbol{\eta}_{\kappa}$ and $\Delta\boldsymbol{\eta}$, the $\frac{1}{2}s^2 + \frac{3}{2}s - 2$ blocks of $\mathbf{w}_2, \dots, \mathbf{w}_s$ belonging to the working space of the current pipelining step and s blocks of the error estimate \mathbf{e} . This leads to a total amount of

$$2n + \left(\frac{1}{2}s^2 + \frac{5}{2}s - 2 \right) B = 2n + \Theta \left(\frac{1}{2}s^2 B \right) \quad (6)$$

vector elements that have to be stored in memory. If no step control is used, the memory space required even reduces to

$$n + \left(\frac{1}{2}s^2 + \frac{3}{2}s - 2 \right) B = n + \Theta \left(\frac{1}{2}s^2 B \right) \quad (7)$$

vector elements, since no elements of $\Delta\boldsymbol{\eta}$ and \mathbf{e} have to be stored.

¹ A similar sweep approach is possible for other L^p norms $\|x\|_p = \left(\sum_{j=1}^n |x_j|^p \right)^{\frac{1}{p}}$.

Table 1. Offsets (in number of blocks) of the overlapping of $\Delta\eta$ with \mathbf{e} and $\mathbf{w}_2, \dots, \mathbf{w}_s$

Offset:	0	s	$s + 3$	\dots	$s + \frac{1}{2}i^2 + \frac{3}{2}i - 2$	\dots	$\frac{1}{2}s^2 + \frac{5}{2}s - 2$
Vector:	$\Delta\eta$	\mathbf{e}	\mathbf{w}_2	\dots	\mathbf{w}_i	\dots	\mathbf{w}_s

One possibility to design an implementation of a low-storage pipelining scheme which supports stepsize control by an embedded solution is the overlapping of $\Delta\eta$ with the vectors \mathbf{e} and $\mathbf{w}_2, \dots, \mathbf{w}_s$ as illustrated in Fig. 11(c). The implementation requires two registers, one of size n which stores the current approximation vector η_κ , and one of size $n + (\frac{1}{2}s^2 + \frac{5}{2}s - 2)B$ which stores $\Delta\eta$ and the elements of \mathbf{e} and $\mathbf{w}_2, \dots, \mathbf{w}_s$ which belong to the working space of the current pipelining step using the offsets shown in Table 1. Using this overlapping the pipelining scheme can be executed without further changes as described in Section 3.2. As the result, the working space of the pipelining steps is embedded into the enlarged register holding $\Delta\eta$ as a compact, consecutive window such that the computation of the next block of $\Delta\eta$ moves this window one block forward.

3.5 Parallel Implementation

The pipelining scheme can exploit parallelism across the ODE system efficiently by making use of the limited access distance of the ODE system to reduce communication costs and to increase the locality of memory references [89]. A parallel low-storage implementation of the pipelining scheme with even higher scalability (cf. Section 5) can easily be obtained with only a few adaptations.

We start with a blockwise data distribution of the $n_B = \lceil n/B \rceil$ blocks such that every processor is responsible for the computation of $\lfloor n_B/p \rfloor$ or $\lceil n_B/p \rceil$ blocks of $\Delta\eta$ and \mathbf{e} , where p is the number of processors. Because in the low-storage implementation the working space of the pipelining steps is embedded into the register holding $\Delta\eta$, it is not possible to use only one global copy of this register to which all participating processors have shared access. Instead, each processor allocates a chunk of memory of size $(\frac{1}{2}s^2 + \frac{5}{2}s - 1)B$ locally. The local allocation can be of advantage on shared-memory systems with non-uniform memory access times (NUMA systems) and enables an implementation for distributed address space. It can, however, be a disadvantage on shared-memory systems with uniform memory access times (UMA systems), because blocks which are required by neighboring processors have to be copied between the local data structures of the threads.

Since the computation of a block J of one argument vector requires the blocks $J - 1$, J , and $J + 1$ of the preceding argument vector, the computation of the first and the last block of a processor's part of an argument vector requires one block of the preceding argument vector that is stored on a neighboring processor. Therefore, communication between neighboring processors is required during the initialization and the finalization of the local pipelines of the processors. In the parallel low-storage implementation, we use a computation order

Table 2. Comparison of data set sizes of different RK integrators occurring in the solution of an ODE system of size $n = 2N^2$ with access distance $d(\mathbf{f}) = 2N$. See explanation in the text.

N	n	I $S(P_7^{2N})$	II $S(n)$	III $S(n + P_7^{2N})$	IV $S(2n)$	V $S(2n + P_7^{2N})$	VI $S((7 + 2)n)$
10^2	$2 \cdot 10^4$	62.5 KB	156.2 KB	218.8 KB	312.5 KB	375.0 KB	1.4 MB
10^3	$2 \cdot 10^6$	625.0 KB	15.3 MB	15.9 MB	30.5 MB	31.1 MB	137.3 MB
10^4	$2 \cdot 10^8$	6.1 MB	1.5 GB	1.5 GB	3.0 GB	3.0 GB	13.4 GB
10^5	$2 \cdot 10^{10}$	61.0 MB	149.0 GB	149.1 GB	298.0 GB	298.1 GB	1.3 TB
10^6	$2 \cdot 10^{12}$	610.4 MB	14.6 TB	14.6 TB	29.1 TB	29.1 TB	131.0 TB

which differs from that of the implementations described in [89], because the embedding of the working space into the register holding $\Delta\boldsymbol{\eta}$ leads to different dependencies. In particular, processors with even processor ID move from lower block indices to higher block indices, while processors with an odd ID move from higher to lower indices. Thus, processors with even ID initialize their pipeline synchronously with their predecessor and finalize their pipeline synchronously with their successor. Similar to the implementations described in [89], the computation pattern used in the initialization and the finalization phase resembles a zipper, and communication can be overlapped with computations.

4 Comparison of Data Set Sizes of Semi-discretized PDEs for Different RK Integrators

The spatial discretization of a d -dimensional PDE system by the method of lines [10] uses a d -dimensional spatial grid. In the following, we assume that the number of grid points in each dimension be equal, and refer to it as N . The ODE system which results from this semi-discretization consists of $n_v \cdot N^d$ equations, where n_v is the number of dependent variables in the PDE system. The spatial derivatives at the grid points are approximated using difference operators which use grid points in a bounded environment of the respective grid point considered. Hence, one spatial dimension can be selected for a subdivision of the ODE system into blocks such that each block represents a N^{d-1} -dimensional slice of the grid and the ODE system has a limited access distance $d(\mathbf{f}) = n_v \cdot r \cdot N^{d-1}$, where r is the maximum distance in the selected dimension between the grid point to which the discretization operator is applied and the grid points used by the discretization operator.

Since the working space of the pipelining scheme requires a storage space of only $\Theta(\frac{1}{2}s^2d(\mathbf{f})) = \Theta(n_v \cdot r \cdot N^{d-1})$ while the system size n grows with $\Theta(N^d)$, the additional storage space needed to store the working space becomes less significant when N increases. As an example, Table 2 shows a comparison of data set sizes for an ODE of size $n = 2N^2$ with access distance $d(\mathbf{f}) = 2N$ such as it may result from a 2D PDE with two dependent variables discretized

using a five-point stencil (e.g., BRUSS2D [1], see also Section 5). The function $S(x)$ used in this table represents the space needed to store x vector elements using double precision (8 bytes per vector element). P_s^B is the number of vector elements in the working space of the pipelining scheme that have to be stored in addition to η_κ (and $\Delta\eta$) if an s -stage method and blocksize B is used, i.e.,

$$P_s^B = \left(\frac{1}{2}s^2 + \frac{5}{2}s - 2 \right) B . \quad (8)$$

Column I shows the size of the working space $S(P_s^B)$ in the case that a 7-stage method is used. We chose $s = 7$ because the popular embedded RK method DOPRI5(4) that we used in the runtime experiments presented in Section 5 has 7 stages. Column II shows the size of a single register holding n vector elements. Columns III and V represent the total amount of memory needed when using the low-storage pipelining implementation with (V) and without (III) stepsize control. In comparison to this, column IV shows the amount of memory needed when using the most space-efficient RK methods that have been proposed in related works, which make use of special coefficient sets and require at least two n -registers. Finally, column VI shows the storage space required when using a conventional embedded RK method with 7 stages and stepsize control or a conventional RK method with 8 stages without stepsize control.

5 Experimental Evaluation

Runtime experiments to evaluate the performance of a sequential and a parallel shared-memory low-storage implementation of the pipelining scheme (PipeDls) in comparison to the pipelining implementation presented in [8,9] (PipeD) and a conventional RK implementation supporting arbitrary right-hand-side functions (D) have been performed on two off-the-shelf SMP servers equipped with Intel Xeon and AMD Opteron quad-core processors and on two of the largest German supercomputer systems, the Jülich Multiprocessor (JUMP) at the Jülich Supercomputing Centre and the High End System in Bavaria 2 (HLRB 2) at the LRZ Munich. JUMP is a cluster system consisting of 14 SMP nodes, each equipped with 32 Power 6 processors which support simultaneous multithreading (SMT). HLRB 2 is an SGI Altix 4700 system. Currently, the system is equipped with 9728 Intel Itanium 2 Montecito processors at 1.6 GHz. The implementations have been developed using C and POSIX Threads for parallelization. They were compiled with GCC 4.3 on the two 4×4 SMP servers and on HLRB 2 and with IBM XL C/C++ V 10.1 on JUMP.

As an example problem we use BRUSS2D [1], a typical example of a PDE discretized by the method of lines. Using an interleaving of the two dependent variables resulting in a mixed-row oriented ordering of the components (BRUSS2D-MIX, cf. [8,9]), this problem has a limited access distance of $d(\mathbf{f}) = 2N$, where the system size is $n = 2N^2$. The experiments presented in the following have been performed using the 7-stage embedded RK method DOPRI5(4) and $\alpha = 2 \cdot 10^{-3}$ as weight of the diffusion term of BRUSS2D.

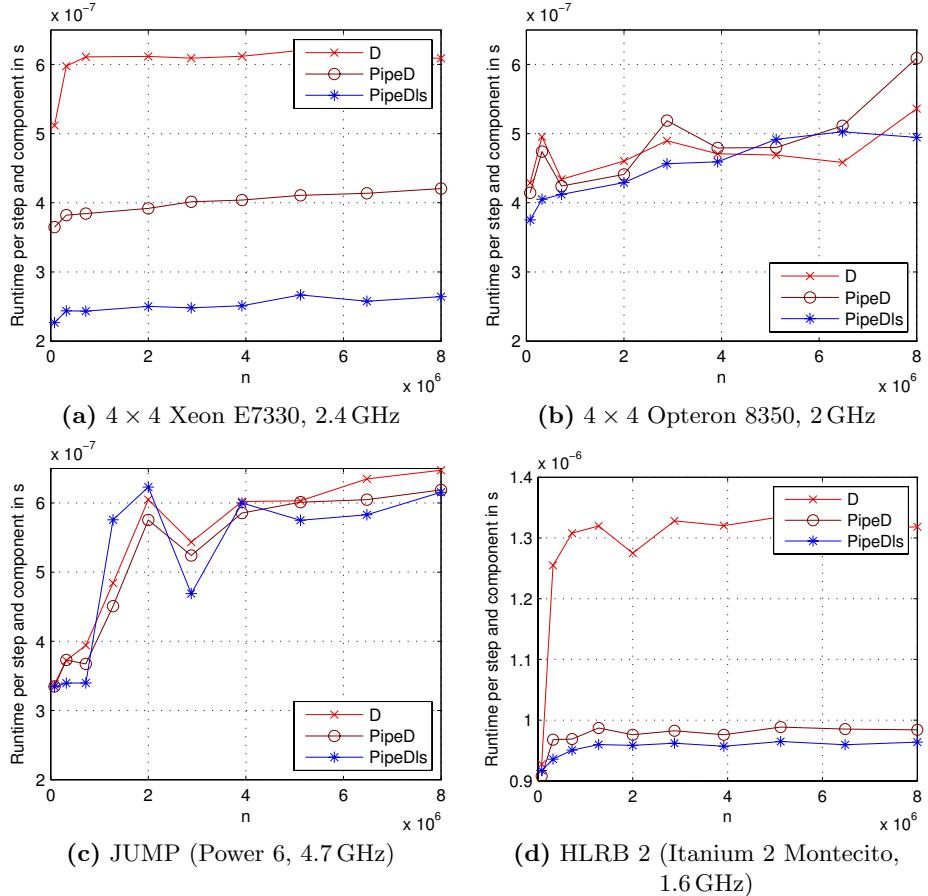


Fig. 2. Sequential execution time per step and per component against system size n

Figure 2 shows a comparison of the sequential runtimes of the 3 implementations on the 4 target systems. On the Opteron processor and on the Power 6 processor the runtimes of the implementations do not differ significantly. But on the Itanium 2 processor the pipelining implementations are considerably faster than the conventional implementation. Moreover, due to its higher locality resulting from the more compact storage of the working space, the low-storage variant of the pipelining scheme runs slightly faster than the implementation presented in [8,9]. The difference in the performance between the two pipelining implementations is even more significant on the Xeon processor.

A comparison of the speedups of the parallel shared-memory implementations on the 4 target systems for selected grid sizes N is shown in Fig. 3. Since our intention is to compare the new low-storage implementation of the pipelining scheme with the pre-existing pipelining implementation from [8,9], we use the

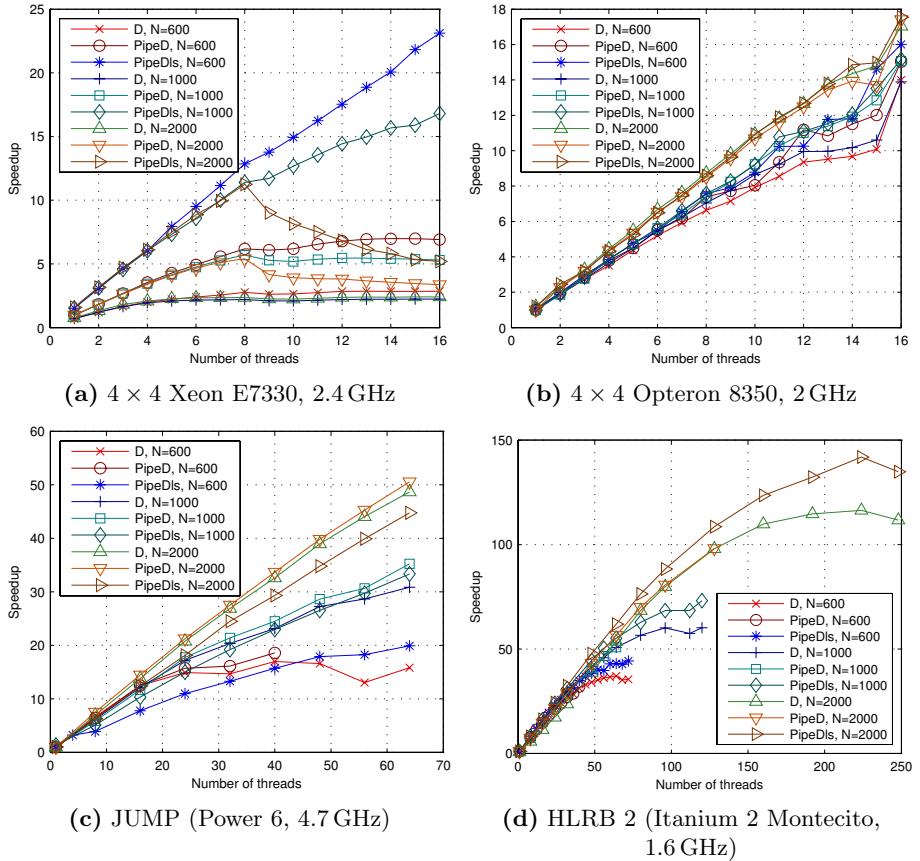


Fig. 3. Speedup against number of threads for selected grid sizes N

sequential execution time of the pre-existing pipelining implementation as the reference for the speedup computation. All 3 implementations scale well for all grid sizes on the 16-core Opteron SMP, where speedups between 14 and 16 have been observed. On JUMP the performance of our current low-storage pipelining implementation is slightly worse than that of the pipelining implementation from [8,9]. We ascribe this to the relatively uniform memory access times on this machine in combination with the fact that the data exchange between neighboring processors is not overlapped by computations. It is remarkable, however, that in most experiments the implementations can take high profit of the SMT capability of the Power 6 processor. On the HLRB 2, where memory access times are substantially more non-uniform, the two pipelining implementations show nearly the same performance. On the Xeon SMP the pipelining implementation from [8,9] clearly outperforms the general implementation, but its maximum speedup still is below 7. The low-storage variant of the pipelining scheme reaches significantly higher speedups between 11 and 23.

6 Conclusions

In this paper, we have proposed an implementation strategy for RK methods which is based on a pipelined processing of the stages of the method. This implementation strategy is suitable for ODE systems with limited access distance. It has been derived from the pipelining scheme presented in [8,9] by an overlapping of vectors. The implementation strategy proposed herein provides a higher locality of memory references and requires less storage space than the implementations presented in [8,9] and other implementations proposed in related works. If no step control is required, only $n + \Theta(\frac{1}{2}s^2d(\mathbf{f}))$ vector elements need to be stored. Efficient step control based on an embedded solution is possible using only one additional n -vector. In contrast to low-storage RK algorithms proposed by other authors, the presented implementation strategy does not impose restrictions on the choice of coefficients of the RK method and can thus be used with popular embedded RK method such as the methods of Dormand and Prince, Verner, Fehlberg and others. Moreover, a parallel implementation of the proposed strategy is possible which has shown a high scalability on four different parallel computer systems.

Acknowledgments. We thank the Jülich Supercomputing Centre and the LRZ Munich for providing access to their supercomputer systems.

References

1. Burrage, K.: *Parallel and Sequential Methods for Ordinary Differential Equations*. Oxford University Press, New York (1995)
2. Kennedy, C.A., Carpenter, M.H., Lewis, R.M.: Low-storage, explicit Runge–Kutta schemes for the compressible Navier–Stokes equations. *Appl. Numer. Math.* 35(3), 177–219 (2000)
3. Calvo, M., Franco, J.M., Rández, L.: Short note: A new minimum storage Runge–Kutta scheme for computational acoustics. *J. Comp. Phys.* 201(1), 1–12 (2004)
4. Berland, J., Bogey, C., Bailly, C.: Optimized explicit schemes: matching and boundary schemes, and 4th-order Runge–Kutta algorithm. In: 10th AIAA/CEAS Aeroacoustics Conference, pp. 1–34 (2004), AIAA Paper 2004-2814
5. Berland, J., Bogey, C., Bailly, C.: Low-dissipation and low-dispersion fourth-order Runge–Kutta algorithm. *Computers & Fluids* 35(10), 1459–1463 (2006)
6. Ruuth, S.J.: Global optimization of explicit strong-stability-preserving Runge–Kutta methods. *Math. Comp.* 75(253), 183–207 (2005)
7. Kennedy, C.A., Carpenter, M.H.: Third-order $2N$ -storage Runge–Kutta schemes with error control. Technical Report NASA TM-109111, National Aeronautics and Space Administration, Langley Research Center, Hampton, VA (1994)
8. Korch, M., Rauber, T.: Scalable parallel RK solvers for ODEs derived by the method of lines. In: Kosch, H., Böszörményi, L., Hellwagner, H. (eds.) *Euro-Par 2003. LNCS*, vol. 2790, pp. 830–839. Springer, Heidelberg (2003)
9. Korch, M., Rauber, T.: Optimizing locality and scalability of embedded Runge–Kutta solvers using block-based pipelining. *J. Par. Distr. Comp.* 66(3), 444–468 (2006)
10. Schiesser, W.E.: *The Numerical Method of Lines*. Academic Press Inc., London (1991)

PSPIKE: A Parallel Hybrid Sparse Linear System Solver[☆]

Murat Manguoglu¹, Ahmed H. Sameh¹, and Olaf Schenk²

¹ Department of Computer Science, Purdue University, West Lafayette IN 47907

² Computer Science Department, University of Basel, Klingelbergstrasse 50, CH-4056 Basel

Abstract. The availability of large-scale computing platforms comprised of tens of thousands of multicore processors motivates the need for the next generation of highly scalable sparse linear system solvers. These solvers must optimize parallel performance, processor (serial) performance, as well as memory requirements, while being robust across broad classes of applications and systems. In this paper, we present a new parallel solver that combines the desirable characteristics of direct methods (robustness) and effective iterative solvers (low computational cost), while alleviating their drawbacks (memory requirements, lack of robustness). Our proposed hybrid solver is based on the general sparse solver PARDISO, and the “Spike” family of hybrid solvers. The resulting algorithm, called PSPIKE, is as robust as direct solvers, more reliable than classical preconditioned Krylov subspace methods, and much more scalable than direct sparse solvers. We support our performance and parallel scalability claims using detailed experimental studies and comparison with direct solvers, as well as classical preconditioned Krylov methods.

Key words: Hybrid Solvers, Direct Solvers, Krylov Subspace Methods, Sparse Linear Systems.

1 Introduction

The emergence of extreme-scale parallel platforms, along with increasing number of cores available in conventional processors pose significant challenges for algorithm and software development. Machines with tens of thousands of processors and beyond place tremendous constraints on the communication requirements of algorithms. This is largely due to the fact that the restricted memory scaling model (generally believed to be no more than linear in the number of processing cores) allows for limited problem scaling. This in turn limits the amount of overhead that can be amortized across useful work in the program.

Increase in the number of cores in a processing unit without a commensurate increase in memory bandwidth, on the other hand, exacerbates an already significant memory bottleneck. Sparse algebra kernels are well-known for their poor

[☆] This work was supported by grants from NSF (NSF-CCF-0635169), DARPA/AFRL (FA8750-06-1-0233), and a gift from Intel and partially supported by the Swiss National Science Foundation under grant 200021 – 117745/1.

processor utilization (typically in the range of 10 - 20% of processor peak). This is a result of limited memory reuse, which renders data caching less effective. Attempts at threading these computations rely on concurrency to trade off memory bandwidth for latency. By allowing multiple outstanding reads, these computations hide high latency of access. However, this is predicated on the ability of the memory system to handle multiple requests concurrently. In view of these emerging hardware trends, it is necessary to develop algorithms and software that strike a more meaningful balance between memory accesses, communication, and computation. Specifically, an algorithm that performs more floating point operations at the expense of reduced memory accesses and communication is likely to yield better performance.

This paper addresses the problem of developing robust, efficient, and scalable sparse linear system solvers. Traditional approaches to linear solvers rely on direct or iterative methods. Direct methods are known to be robust – i.e., they are capable of handling relatively ill-conditioned linear systems. However, their memory requirements and operation counts are typically higher. Furthermore, their scaling characteristics on very large numbers of processors while maintaining robustness, remain unresolved. Iterative methods, on the other hand, have lower operation counts and memory requirements, but are not as robust.

In this paper, we present a new parallel solver that incorporates the desirable characteristics of direct and iterative solvers while alleviating their drawbacks. Specifically, it is robust in the face of ill-conditioning, has lower memory references, and is amenable to highly efficient parallel implementation.

We first discuss the basic Spike solver for banded systems and PSPIKE (also known as Spike-PARDISO) hybrid solver for general sparse linear systems. Next, we demonstrate the scalability and performance of our solver on up to 256 cores of a distributed memory platform for a variety of applications. Finally, we enhance the robustness of our method through symmetric and nonsymmetric weighted reordering. We compare our methods to commonly used incomplete LU (ILU) factorization-based preconditioners, and direct solvers to demonstrate superior performance characteristics of our solver. We would like to note that there has been several attempts to design hybrid algorithms in the past based on either classical LU factorization or the schur complement method. Our approach, however, is different and exhibits superior scalability.

The test platform for our experiments is a cluster of dual quad-core Intel Xeon E5462 nodes, with each core operating at 2.8 GHz. The nodes are interconnected via Infiniband.

2 The Basic Spike Algorithm

Let $Ax = f$ be a nonsymmetric diagonally dominant system of linear equations where A is of order n and bandwidth $2m + 1$. Unlike classical banded solvers such as those in LAPACK that are based on LU-factorization of A , the spike algorithm [1234567] is based on the factorization $A = D \times S$, where D is a block-diagonal matrix and S is the spike matrix shown in Figure 1 for three

$$A = D \times S$$

Fig. 1. Spike decomposition where $A = D * S, S = D^{-1}A, B_j, C_j \in \mathbb{R}^{m \times m}$

partitions. Note that the block diagonal matrices A_1, A_2 , and A_3 are nonsingular by virtue of the diagonal dominance of A . For the example in Figure 1, the basic Spike algorithm consists of the following stages:

- **Stage 1:** Obtain the LU-factorization (without pivoting) of the diagonal blocks A_j (i.e. $A_j = L_j U_j, j = 1, 2, 3$)
- **Stage 2:** Forming the spike matrix S and updating the right hand side
 - (i) solve $L_1 U_1 [V_1, g_1] = [(\begin{smallmatrix} 0 \\ B_1 \end{smallmatrix}), f_1]$
 - (ii) solve $L_2 U_2 [W_2, V_2, g_2] = [(\begin{smallmatrix} C_2 \\ 0 \end{smallmatrix}), (\begin{smallmatrix} 0 \\ B_2 \end{smallmatrix}), f_2]$
 - (iii) solve $L_3 U_3 [W_3, V_3, g_3] = [(\begin{smallmatrix} C_3 \\ 0 \end{smallmatrix}), f_3]$
- $f_j, i \leq j \leq 3$, are the corresponding partitions of the right hand side f .
- **Stage 3:** Solving the reduced system,

$$\begin{bmatrix} I & V_1^{(b)} & 0 & 0 \\ W_2^{(t)} & I & 0 & V_2^{(t)} \\ W_2^{(b)} & 0 & I & V_2^{(b)} \\ 0 & 0 & W_3^{(t)} & I \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{bmatrix} = \begin{bmatrix} g_1^{(b)} \\ g_2^{(t)} \\ g_2^{(b)} \\ g_3^{(t)} \end{bmatrix} \quad (1)$$

where $(V_i^{(b)}, W_i^{(b)})$ and $(V_i^{(t)}, W_i^{(t)})$ are the bottom and top $m \times m$ blocks of (V_i, W_i) , respectively. Similarly, $g_i^{(b)}$ and $g_i^{(t)}$ are the bottom and top m elements of each g_i . Once this smaller reduced system is solved the three partitions $x_i, i = 1, 2, 3$, of the solution x are obtained as follows:

- (i) $x_1 = g_1 - V_1 z_2$
- (ii) $x_2 = g_2 - W_2 z_1 - V_2 z_4$
- (iii) $x_3 = g_3 - W_3 z_3$

Clearly, the above basic scheme may be made more efficient if in stage 2 one can generate only the bottom and top $m \times m$ tips of the spikes V_i and W_i , as well as the corresponding bottom and top tips of g_i . In this case, once the reduced system is solved, solving the system $Ax = f$ is reduced to solving the independent systems:

- (i) $L_1 U_1 x_1 = f_1 - B_1 z_2$
- (ii) $L_2 U_2 x_2 = f_2 - C_2 z_1 - B_2 z_4$
- (iii) $L_3 U_3 x_3 = f_3 - C_3 z_3$

If the matrix A is not diagonally dominant, we cannot guarantee that the diagonal blocks, A_i , are nonsingular. However, if we obtain the LU-factorization, without pivoting, of each block A_i using diagonal boosting (perturbation), then

$$L_i U_i = (A_i + \delta A_i) \quad (2)$$

in which $\|\delta A_i\| = \mathcal{O}(\epsilon \|A_i\|)$, where ϵ is the unit roundoff. In this case, we will need to solve $Ax = f$ using an outer iterative scheme with the preconditioner being the matrix M that is identical to A , except that each diagonal block A_i is replaced by $L_i U_i$ in Equation 2. These systems, of the form $My = r$, are solved using the Spike scheme outlined above.

3 The PSPIKE Scheme

PARDISO [89] is a state-of-the-art direct sparse linear system solver. Its performance and robustness have been demonstrated in the context of several applications. The hybrid PSPIKE scheme can be used for solving general sparse systems as follows. If the elements of the coefficient matrix naturally decay as one moves away from the main diagonal we do not need to reorder the linear system. Otherwise, the sparse matrix A is reordered via a nonsymmetric reordering scheme, which ensures that none of the diagonal elements are zero, followed by a weighted reordering scheme that attempts to move as many of the largest elements as possible to within a narrow central band, M . This banded matrix, M is used as a preconditioner for an outer iterative scheme, e.g., BiCGStab [10], for solving $Ax = f$. The major operations in each iteration are: (i) matrix-vector products, and (ii) solving systems of the form $My = r$.

Solving systems $My = r$ involving the preconditioner is accomplished using our proposed algorithm: PSPIKE as follows: the LU-factorization of each diagonal block partition M_i (banded and sparse within the band) is obtained using PARDISO with supernodal pivoting and weighted graph matchings. The most recent version of PARDISO is also capable of obtaining the top and bottom tips of the left and right spikes \hat{W}_i and \hat{V}_i , as well as the corresponding tips of the updated right hand side subvectors. Further, having the largest elements within the band, whether induced by the reordering or occurring naturally, allows us to approximate (or truncate) the resulting reduced system by its block diagonal, $\hat{S} = \text{diag}(\hat{S}_1, \hat{S}_2, \dots, \hat{S}_p)$, where p is the number of partitions and,

$$\hat{S}_j = \begin{bmatrix} I & \hat{V}_j^{(b)} \\ \hat{W}_{j+1}^{(t)} & I \end{bmatrix}. \quad (3)$$

This also enhances concurrency, especially when M has a large bandwidth. For a more detailed discussion of the decay rate of spikes and the truncated Spike algorithm we refer the reader to [11]. In the following section, we will demonstrate the

suitability of the PSPIKE solver for implementation on clusters consisting of several nodes in which each node is a multicore processor. While PARDISO is primarily suitable for single node platforms, PSPIKE is scalable across multiple nodes. We would also like to mention that our approach is suitable for MPI/OpenMP hybrid parallelism where each node can use the threading capability of PARDISO, as well as pure MPI parallelism where no threads are used. We give example of both programming paradigms in the following sections. The number of partitions in our implementation is the same as the number of MPI processes.

4 A Weighted Reordering Scheme

The decay of the absolute value of the matrix elements as one moves away from the main diagonal is essential for the convergence of the PSPIKE scheme. There are, however, linear systems which do not immediately possess this property. In fact the same system may or may not have decaying elements based on the reordering. We propose an additional layer of weighted symmetric/nonsymmetric reordering in order to enhance the robustness of the PSPIKE scheme for such systems. We will call this preprocessing method as the Weighted Reordering method(WR) in the rest of the paper.

Given a linear system $Ax = f$, we first apply a nonsymmetric row permutation as follows: $QAx = Qf$. Here, Q is the row permutation matrix that either maximizes the number of nonzeros on the diagonal of A [12], or the permutation that maximizes the product of the absolute values of the diagonal entries [13]. The first algorithm is known as the scheme for the *maximum traversal search*. Both algorithms are implemented in the MC64 subroutine of the HSL [14] library.

Following the above nonsymmetric reordering and optional scaling, we apply the symmetric permutation P as follows:

$$(PQAP^T)(Px) = (PQf). \quad (4)$$

We use HSL-MC73 [15] weighted spectral reordering to obtain the symmetric permutation P that minimizes the bandwidth encapsulating a specified fraction of the total magnitude of nonzeros in the matrix. This permutation is determined from the symmetrized matrix $|A| + |A^T|$.

In the following sections the preconditioner we extract is treated as either dense or sparse within the band.

5 Banded Preconditioners (Dense within the Band)

In this section we study the convergence characteristics of various solvers on a uniprocessor. This set of problems consists of symmetric and nonsymmetric general sparse matrices from the University of Florida [16]. For each matrix we generated the corresponding right hand-side using a solution vector of all ones to ensure that $f \in \text{span}(A)$. The number k represents the bandwidth of the preconditioner detected by our method. The linear systems involving the

Table 1. Properties of Test Matrices

Matrix	k	Dimension(N)	Non-zeros(nnz)	Type
1.FINAN512	50	74,752	596,992	Financial Optimization
2.FEM_3D_THERMAL1	50	17,880	430,740	3D Thermal FEM
3.RAJAT31	30	4,690,002	20,316,253	Circuit Simulation
4.H2O	50	67,024	2,216,736	Quantum Chemistry
5.APPU	50	14,000	1,853,104	NASA Benchmark
6.BUNDLE1	50	10,581	770,811	3D Computer Vision
7.RAJAT30	30	643,994	6,175,244	Circuit Simulation
8.DW8191	182	8,192	41,746	Dielectric Waveguide
9.DC1	50	116,835	766,396	Circuit Simulation
10.FP	2	7,548	834,222	Electromagnetics
11.PRE2	30	659,033	5,959,282	Harmonic Balance Method
12.KKT_POWER	30	2,063,494	14,612,663	Nonlinear Optimization
13.RAEFSKY4	50	19,779	1,328,611	Structural Mechanics
14.ASIC_680k	3	682,862	3,871,773	Circuit Simulation
15.2D_54019_HIGHK	2	54,019	996,414	Device Simulation

preconditioner are solved via LAPACK. After obtaining the reordered system via WR, we determine the half-bandwidth of the preconditioner (k) such that 99.99% of the total weight of the matrix is encapsulated within the band. If $n > 10,000$ and $n > 500,000$ we enforce upper limits of 50 and 30, respectively for k . For establishing a competitive baseline, we obtain an ILUT preconditioner via the maximum product traversal and scaling proposed by Benzi et al. [7] (ILUTI). Furthermore, we use PARDISO with the METIS [18] reordering and enable the nonsymmetric ordering option for indefinite systems. For ILUPACK [19], we use

Table 2. Total solve time for ILUTI, WR, ILUPACK, PARDISO and RCM methods

Matrix	Condest	Banded		LU based		
		WR	RCM	ILUTI(*, 10^{-1})	PARDISO	ILUPACK
1	9.8×10^1	0.87	0.29	0.14	1.28	0.5
2	1.7×10^3	0.41	0.19	0.19	1.48	0.23
3	4.4×10^3	457.2	F	193.5	91.6	F
4	4.9×10^3	5.63	4.89	0.84	450.73	2.86
5	1.0×10^4	0.82	0.51	0.88	270.17	44.6
6	1.3×10^4	0.7	0.14	0.27	0.88	0.27
7	8.7×10^4	205.4	F	459.1	7.6	F
8	1.5×10^7	0.33	0.07	F	0.78	F
9	1.1×10^{10}	1.95	8.99	15.8	3.14	2.07
10	8.2×10^{12}	0.42	0.04	F	1.74	0.46
11	3.3×10^{13}	7.0	F	F	45.3	F
12	4.2×10^{13}	F	F	F	449.1	F
13	1.5×10^{14}	0.27	0.09	0.59	1.28	F
14	9.4×10^{19}	2.0	F	239.0	27.2	F
15	8.1×10^{32}	0.21	0.06	0.11	0.95	1.44

the PQ reordering option and a drop tolerance of 10^{-1} with a bound on the condition number parameter 50 (recommended in the user documentation for general problems). In addition, we enable options in ILUPACK to use matchings and scalings (similar to MC64). The BiCGStab iterations for solving systems are terminated when $\|\hat{r}_k\|_\infty / \|\hat{r}_0\|_\infty < 10^{-5}$.

In Table 2, we show the total solution time of each algorithm including the direct solver PARDISO. A failure indicated by F represent the method either failed during the factorization or iterative solution stages. We note that incomplete LU based methods (ILUTI and ILUPACK) fails in more cases than WR. While PARDISO is the most robust method for the set of problems, in terms of total solve time it is faster than WR only in two cases. Incomplete LU factorization based preconditioners are faster than banded preconditioners(WR and RCM) for problems that are well conditioned (condest $\leq 3.3 \times 10^4$).

6 PSPIKE for Solving General Sparse Linear Systems (Sparse within the Band)

We apply the PSPIKE scheme on four large general sparse systems (Table 3). Two of the matrices are obtained from the UF Sparse matrix collection, while the other two are from a nonlinear optimization package [20]. The timings we report

Table 3. Selection of Test Matrices

Name	Application	unknowns	nonzeros	symmetry
MIPS50S1	Optimization	235, 592	1, 009, 736	symmetric
MIPS80S1	Optimization	986, 552	4, 308, 416	symmetric
G3_CIRCUIT	Circuit Simulation	1, 585, 478	4, 623, 152	symmetric
CAGE14	DNA Model	1, 505, 785	27, 130, 349	general

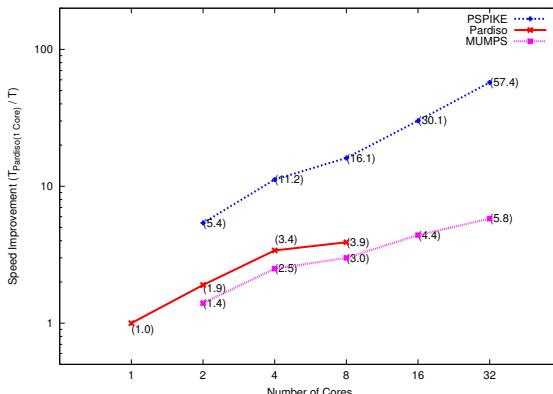


Fig. 2. MIPS50S1: The speed improvement compared to PARDISO using one core (70.4 seconds)

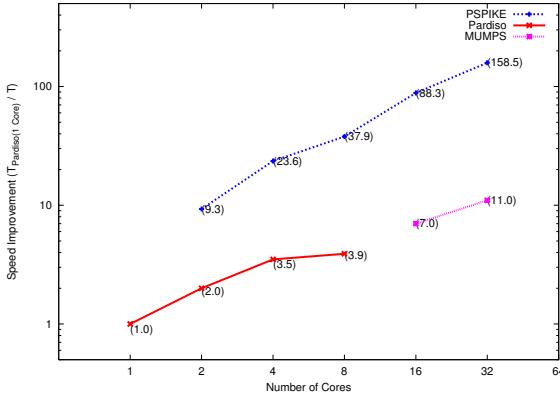


Fig. 3. MIPS80S1: The speed improvement compared to PARDISO using one core (1, 460.2 seconds)

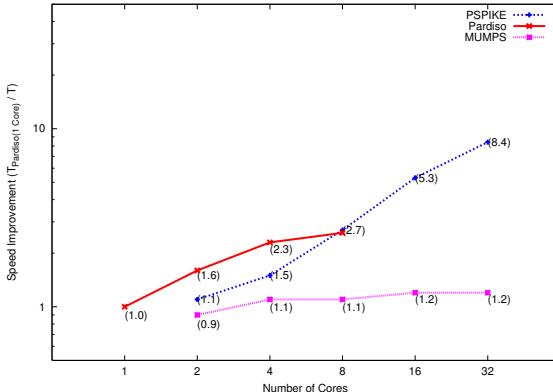


Fig. 4. G3_CIRCUIT: The speed improvement compared to PARDISO using one core (73.5 seconds)

are total times, including reordering, symbolic factorization, factorization, and solution. In these tests, we use METIS reordering for both MUMPS [21][22][23] and PARDISO. For PSPIKE, We use the same number of MPI processes as the number of cores and 1 thread per MPI process. We use BiCGStab with a banded preconditioner to solve these systems where the bandwidth of the preconditioner is 101. We apply HSL's MC64 reordering to maximize the absolute value of the products of elements on the main diagonal [13]. The BiCGStab iterations for solving systems are terminated when $\|\hat{r}_k\|_\infty / \|\hat{r}_0\|_\infty < 10^{-5}$. We note that the reason we use BiCGStab for the symmetric systems is that the coefficient matrix becomes nonsymmetric after using the nonsymmetric reordering (MC64). In Figures 2 and 3, we show the speed improvement realized by the PSPIKE scheme compared to the uniprocessor time spent in PARDISO for two nonlinear optimization problems. We note that the speed improvement of PSPIKE is

Table 4. Total Solve Time for CAGE14 using PSPIKE

Number of Cores	2	4	8	16	32
Time(seconds)	35.3	30.9	21.7	8.7	3.9

enhanced as the problem size increases. Furthermore, MUMPS runs out of memory for the larger problem if less than 32 cores (2 nodes) are used.

Figure 4 shows the speed improvement for the circuit simulation problem. Unlike the other two problems, MUMPS spends significant portion of the time in the reordering/symbolic factorization stage, which is not scalable. CAGE14 has the largest number of nonzeros per row among the four problems. As a result, both MUMPS and PARDISO run out of memory due to large fillin. The PSPIKE scheme, on the other hand, can solve this system. In Table 4 we present the total solve times for PSPIKE. The superlinear speed improvement we observe is due to the fact that as smaller blocks are factored via PARDISO, the fill-in improves much faster than the reduction in the matrix dimension.

7 PSPIKE for Solving Linear Systems Arising in a PDE-Constrained Optimization Problem (Sparse within the Band)

In this section we propose a block factorization based scheme for solving linear systems that arise in a PDE-constrained optimization problem [24]. In which the inner linear solves are accomplished by PSPIKE. Linear systems extracted from a nonlinear solve have the following block structure:

$$\begin{bmatrix} D & B^T \\ H & C^T \\ B & C & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \end{bmatrix} \quad (5)$$

Here, $D \in \mathbb{R}^{n \times n}$ is diagonal and $D_{ii} > 0$ for $i = 1, 2, \dots, n$. Furthermore $H \in \mathbb{R}^{k \times k}$ is symmetric positive definite and $C \in \mathbb{R}^{k \times n}$ is dense with $k \ll n$. $B \in \mathbb{R}^{n \times n}$ is nonsymmetric banded and sparse within the band.

Premultiplying the above equation by $\begin{bmatrix} D^{-1} & & \\ & H^{-1} & \\ & & I \end{bmatrix}$ we get

$$\begin{bmatrix} I & \tilde{B}^T \\ I & \tilde{C}^T \\ B & C & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} \hat{f}_1 \\ \hat{f}_2 \\ f_3 \end{bmatrix} \quad (6)$$

where $\tilde{B}^T = D^{-1}B^T$, $\tilde{C}^T = H^{-1}C^T$, $\hat{f}_1 = D^{-1}f_1$ and $\hat{f}_2 = H^{-1}f_2$. Rearranging the rows and columns, we have the system,

$$\begin{bmatrix} I & \tilde{B}^T & 0 \\ B & 0 & C \\ 0 & \tilde{C}^T & I \end{bmatrix} \begin{bmatrix} x_1 \\ x_3 \\ x_2 \end{bmatrix} = \begin{bmatrix} \hat{f}_1 \\ f_3 \\ \hat{f}_2 \end{bmatrix} \quad (7)$$

Observing that

$$\begin{bmatrix} I & \tilde{B}^T \\ B & 0 \end{bmatrix}^{-1} = \begin{bmatrix} 0 & B^{-1} \\ \tilde{B}^{-T} & -\tilde{B}^{-T}B^{-1} \end{bmatrix} \quad (8)$$

we can see that x_2 can be obtained by solving the small system,

$$(H + J^T D J)x_2 = (f_2 - J^T f_1 + J^T D b) \quad (9)$$

in where J and b are obtained by solving $B J = C$ and $B b = f_3$, respectively. Consequently, x_1 and x_2 can be computed via $x_1 = b - J x_2$ and $x_3 = B^{-T}(f_1 - D x_1)$. The solution process requires solving linear systems with the coefficient matrices B^T and B . We use BiCGStab with a banded preconditioner to solve these systems, in which the systems involving the preconditioner are solved via the PSPIKE scheme.

The BiCGStab iterations for solving systems involving B^T and B are terminated when $\|\hat{r}_k\|_\infty / \|\hat{r}_0\|_\infty < \epsilon_{in}$, where \hat{r}_0 and \hat{r}_k correspond to the initial residual and the residual at k^{th} iteration, respectively, and the systems involving the preconditioners are solved via the PSPIKE scheme.

We considered linear systems extracted from the first, tenth (middle), and twenty first (last) iterations. However, since the results are uniform across these three systems, we present in Figure 5 the results for the linear system of the tenth Newton iteration. Matrix dimensions are $n = 702, 907, k = 23$ and the bandwidth of the preconditioner is 21. Figure 5 illustrates the speed improvement for an MPI/OpenMP hybrid implementation (8 cores(threads) per MPI processes) of the PSPIKE scheme. We demonstrate the speed improvement of the PSPIKE scheme for two stopping criteria, 10^{-5} and 10^{-7} . The corresponding final relative residuals are $\mathcal{O}(10^{-5})$ and $\mathcal{O}(10^{-7})$, respectively. For PARDISO and MUMPS the final relative residuals are $\mathcal{O}(10^{-12})$.

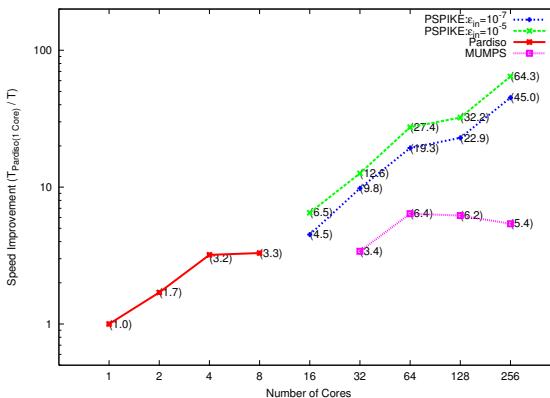


Fig. 5. The speed improvement compared to PARDISO using one core (1338 seconds)

8 Conclusions

We have demonstrated that our PSPIKE solver is highly efficient and scalable on multicore/message passing platforms. We also introduce the notion of weighted symmetric/nonsymmetric reordering to further enhance the robustness of the PSPIKE scheme. We demonstrate this enhanced robustness using a wide variety of test matrices.

Acknowledgments. We thank Ananth Grama and Mehmet Koyuturk for providing WR method and many useful comments, and Patrick Amestoy for recommendations that helped us in using the parallel direct solver MUMPS.

References

1. Chen, S.C., Kuck, D.J., Sameh, A.H.: Practical parallel band triangular system solvers. *ACM Transactions on Mathematical Software* 4(3), 270–277 (1978)
2. Lawrie, D.H., Sameh, A.H.: The computation and communication complexity of a parallel banded system solver. *ACM Trans. Math. Softw.* 10(2), 185–195 (1984)
3. Berry, M.W., Sameh, A.: Multiprocessor schemes for solving block tridiagonal linear systems. *The International Journal of Supercomputer Applications* 1(3), 37–57 (1988)
4. Dongarra, J.J., Sameh, A.H.: On some parallel banded system solvers. *Parallel Computing* 1(3), 223–235 (1984)
5. Polizzi, E., Sameh, A.H.: A parallel hybrid banded system solver: the spike algorithm. *Parallel Comput.* 32(2), 177–194 (2006)
6. Polizzi, E., Sameh, A.H.: Spike: A parallel environment for solving banded linear systems. *Computers & Fluids* 36(1), 113–120 (2007)
7. Sameh, A.H., Sarin, V.: Hybrid parallel linear system solvers. *Inter. J. of Comp. Fluid Dynamics* 12, 213–223 (1999)
8. Schenk, O., Gärtner, K.: Solving unsymmetric sparse systems of linear equations with pardiso. *Future Generation Computer Systems* 20(3), 475–487 (2004) Selected numerical algorithms
9. Schenk, O., Gärtner, K.: On fast factorization pivoting methods for sparse symmetric indefinite systems. *Electronic Transactions on Numerical Analysis* 23, 158–179 (2006)
10. van der Vorst, H.A.: Bi-cgstab: a fast and smoothly converging variant of bi-cg for the solution of nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.* 13(2), 631–644 (1992)
11. Mikkelsen, C.C.K., Manguoglu, M.: Analysis of the truncated spike algorithm. *SIAM Journal on Matrix Analysis and Applications* 30(4), 1500–1519 (2008)
12. Duff, I.S.: Algorithm 575: Permutations for a zero-free diagonal [f1]. *ACM Trans. Math. Softw.* 7(3), 387–390 (1981)
13. Duff, I.S., Koster, J.: The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. *SIAM Journal on Matrix Analysis and Applications* 20(4), 889–901 (1999)
14. HSL: A collection of Fortran codes for large-scale scientific computation (2004), <http://www.cse.scitech.ac.uk/nag/hsl/>

15. Hu, Y., Scott, J.: HSL_MCT3: a fast multilevel Fiedler and profile reduction code. Technical Report RAL-TR-2003-036 (2003)
16. Davis, T.A.: University of Florida sparse matrix collection. NA Digest (1997)
17. Benzi, M., Haws, J.C., Tuma, M.: Preconditioning highly indefinite and nonsymmetric matrices. SIAM J. Sci. Comput. 22(4), 1333–1353 (2000)
18. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. 20(1), 359–392 (1998)
19. Bollhöfer, M., Saad, Y., Schenk, O.: ILUPACK Volume 2.1—Preconditioning Software Package (May 2006), <http://ilupack.tu-bs.de>
20. Wächter, A., Biegler, L.T.: On the implementation of a primal-dual interior point filter line search algorithm for large-scale nonlinear programming. Mathematical Programming 106(1), 25–57 (2006)
21. Amestoy, P.R., Guermouche, A., L'Excellent, J.Y., Pralet, S.: Hybrid scheduling for the parallel solution of linear systems. Parallel Comput. 32(2), 136–156 (2006)
22. Amestoy, P.R., Duff, I.S., L'Excellent, J.Y., Koster, J.: A fully asynchronous multifrontal solver using distributed dynamic scheduling. SIAM J. Matrix Anal. Appl. 23(1), 15–41 (2001)
23. Amestoy, P.R., Duff, I.S.: Multifrontal parallel distributed symmetric and unsymmetric solvers. Comput. Methods Appl. Mech. Eng. 184, 501–520 (2000)
24. Schenk, O., Manguoglu, M., Sameh, A., Christian, M., Sathe, M.: Parallel scalable PDE-constrained optimization: antenna identification in hyperthermia cancer treatment planning. Computer Science - Research and Development 23(3), 177–183 (2009)

Out-of-Core Computation of the QR Factorization on Multi-core Processors

Mercedes Marqués¹, Gregorio Quintana-Ortí¹, Enrique S. Quintana-Ortí¹,
and Robert van de Geijn²

¹ Depto. de Ingeniería y Ciencia de Computadores, Universidad Jaume I (UJI),
12.071–Castellón, Spain

{mmarques, gquintan, quintana}@icc.uji.es

² Department of Computer Sciences, The University of Texas at Austin,
Austin, TX 78712
rvdg@cs.utexas.edu

Abstract. We target the development of high-performance algorithms for dense matrix operations where data resides on disk and has to be explicitly moved in and out of the main memory. We provide strong evidence that, even for a complex operation like the QR factorization, the use of a run-time system creates a separation of concerns between the matrix computations and I/O operations with the result that no significant changes need to be introduced to existing in-core algorithms. The library developer can thus focus on the design of algorithms-by-blocks, addressing disk memory as just another level of the memory hierarchy. Experimental results for the out-of-core computation of the QR factorization on a multi-core processor reveal the potential of this approach.

Keywords: Dense linear algebra, out-of-core computation, QR factorization, multi-core processors, high performance.

1 Introduction

Practical efforts to solve very large dense linear systems employ message-passing libraries on distributed-memory systems, extending the memory hierarchy to include secondary memory; see, e.g., [1234]. However, the constant evolution of computer architectures and, more recently, the uprise of general-purpose multi-core processors and hardware accelerators (Cell B.E., GPUs, etc.) is changing the scale of what is considered a *large* problem.

In a previous paper [5] we employed a simple operation like the Cholesky factorization to introduce a high-level approach for computing out-of-core (OOC) dense matrix operations on multi-core processors. We showed there that a matrix with 100,000 rows and columns can be factorized in less than one hour using an eight-core processor. (This problem would have been solved using a distributed-memory platform just a couple of years ago.) Key to our approach is a run-time system which deals with I/O from/to disk, implements a software cache, and overlaps I/O with computation. The most remarkable property, however, is that no significant change is needed to the in-core `libflame` library code [6].

The major contribution of this paper is to provide much stronger and practical evidence of the validity of our approach, analyzing the programmability and performance issues in detail using a more complex operation, the QR factorization. We will demonstrate that the run-time completely hides secondary memory to the library developer, who can focus on developing and parallelizing algorithms-by-blocks (also called tiled algorithms; see, e.g., [4]), with a notable increase in the programmer's productivity.

Our results also show that, provided the user is willing to wait a few hours for the answer, the approach may become a highly cost-effective solution for most dense matrix operations, making it possible that projects with moderate budgets address problems of moderate scale (dimensions of $O(10,000 - 100,000)$) using multi-core processors. Examples of problems that require the solution of large dense linear systems or linear least-squares problems of this dimension include the estimation of Earth's gravitational field, boundary element formulations in electromagnetism and acoustics, and molecular dynamics simulations [7,8,9,10,11].

The rest of the paper is structured as follows. In Section 2 we describe the tiled left-looking algorithm for computing an OOC QR factorization proposed in [12]. Algorithms for this operation are presented using the FLAME notation there, and the multi-threaded parallelization of the basic building kernels is also analyzed in that section. The traditional OOC implementation of the tiled QR factorization and the new run-time are described in Section 3. Finally, experimental results on a multi-core processor with two Intel Xeon QuadCore processors are reported in Section 4, and concluding remarks follow in Section 5.

2 A Tiled Algorithm for the QR Factorization

The QR factorization of a matrix $A \in \mathbb{R}^{m \times n}$ decomposes this matrix into the product

$$A = QR,$$

where $Q \in \mathbb{R}^{m \times m}$ is orthogonal and $R \in \mathbb{R}^{m \times n}$ is upper triangular.

Traditional in-core algorithms for the QR factorization employ Householder reflectors [13] to annihilate the subdiagonal elements of the matrix, processing one column per iteration (from left to right), and effectively reducing A to the upper triangular factor R . In practice, the elements of R overwrite the corresponding entries of A and Q is not formed explicitly; instead, the reflectors are stored in compact form using the strictly lower triangle of the matrix (plus some negligible work space). Blocked algorithms build upon this procedure to improve data locality: at each iteration, the current panel (or slab) of columns is factored, and the columns to its right are updated using efficient level-3 *Basic Linear Algebra Subprograms* (BLAS) [4].

2.1 The Tiled Left-Looking Algorithm

While right-looking blocked algorithms which proceed by slabs in general yield high performance for in-core operations, tiled left-looking algorithms are usually

Algorithm: $[A] := \text{QR_B}(A)$ Partition $A \rightarrow (A_L A_R)$ where A_L is 0 tiles wide while $n(A_L) < n(A)$ do Repartition $(A_L A_R) \rightarrow (A_0 A_1 A_2)$ where A_1 is 1 tile wide <hr/> $A_1 := \text{QR_B1}(A_0, A_1)$ <hr/> Continue with $(A_L A_R) \leftarrow (A_0 A_1 A_2)$ endwhile	Algorithm: $[B] := \text{QR_B1}(A, B)$ Partition $A \rightarrow \begin{pmatrix} A_{TL} & A_{TR} \\ A_{BL} & A_{BR} \end{pmatrix}, B \rightarrow \begin{pmatrix} B_T \\ B_B \end{pmatrix}$ where A_{TL} has 0×0 tiles, B_T is 0 tiles high while $n(A_{TL}) < n(A)$ do Repartition $\begin{pmatrix} A_{TL} & A_{TR} \\ A_{BL} & A_{BR} \end{pmatrix} \rightarrow \begin{pmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{pmatrix},$ $\begin{pmatrix} B_T \\ B_B \end{pmatrix} \rightarrow \begin{pmatrix} B_0 \\ B_1 \\ B_2 \end{pmatrix}$ where A_{11} and B_1 are tiles <hr/> $B_1 := \text{APPLY_Q}(A_{11}, B_1)$ $[B_1, B_2] := \text{QR_B2}(A_{11}, B_1, A_{21}, B_2)$ $B_1 := \text{QR}(B_1)$ $[B_1, B_2] := \text{QR_B3}(B_1, B_2)$ <hr/> Continue with $\begin{pmatrix} A_{TL} & A_{TR} \\ A_{BL} & A_{BR} \end{pmatrix} \leftarrow \begin{pmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{pmatrix},$ $\begin{pmatrix} B_T \\ B_B \end{pmatrix} \leftarrow \begin{pmatrix} B_0 \\ B_1 \\ B_2 \end{pmatrix}$ endwhile
---	--

Fig. 1. Tiled algorithms for computing the QR factorization

preferred for OOC computations due to their higher scalability and reduced I/O. We next describe the tiled left-looking algorithm for the QR factorization introduced in [12].

Consider a partitioning of the matrix $A \in \mathbb{R}^{m \times n}$ into square tiles of size t , $A \rightarrow (A_{ij}) \in \mathbb{R}^{t \times t}$. (For simplicity, we assume that both the row and column dimensions of the matrix are integer multiples of t .) Figures 1 and 2 show the tiled algorithm QR_B for computing the QR factorization of this matrix using the FLAME notation [13]. There $m(X)/n(X)$ stand for the number of row/column tiles of a matrix X . We believe the rest of the notation is intuitive.

FLAME also comprises a set of high-level applications programming interfaces (APIs) which allow to easily code algorithms in FLAME notation [15]. Using the Spark web site (<http://www.cs.utexas.edu/users/flame>) C codes for the algorithms in Figures 1 and 2 can be obtained in a matter of minutes.

Algorithm: $[C, D] := \text{QR_B2}(A, B, C, D)$

Partition $C \rightarrow \left(\begin{array}{c} C_T \\ C_B \end{array} \right)$, $D \rightarrow \left(\begin{array}{c} D_T \\ D_B \end{array} \right)$
where C_T and D_T are 0 tiles high

while $m(C_T) < m(C)$ **do**
Repartition
 $\left(\begin{array}{c} C_T \\ C_B \end{array} \right) \rightarrow \left(\begin{array}{c} C_0 \\ C_1 \\ C_2 \end{array} \right)$, $\left(\begin{array}{c} D_T \\ D_B \end{array} \right) \rightarrow \left(\begin{array}{c} D_0 \\ D_1 \\ D_2 \end{array} \right)$
where C_1 and D_1 are tiles

$$\left(\begin{array}{c} B \\ D_1 \end{array} \right) := \text{APPLY_QTD} \left(\left(\begin{array}{c} A \\ C_1 \end{array} \right), \left(\begin{array}{c} B \\ D_1 \end{array} \right) \right)$$

Continue with
 $\left(\begin{array}{c} C_T \\ C_B \end{array} \right) \leftarrow \left(\begin{array}{c} C_0 \\ C_1 \\ C_2 \end{array} \right)$, $\left(\begin{array}{c} D_T \\ D_B \end{array} \right) \leftarrow \left(\begin{array}{c} D_0 \\ D_1 \\ D_2 \end{array} \right)$

endwhile

Algorithm: $[A, C] := \text{QR_B3}(A, C)$

Partition $C \rightarrow \left(\begin{array}{c} C_T \\ C_B \end{array} \right)$
where C_T is 0 tiles high

while $m(C_T) < m(C)$ **do**
Repartition
 $\left(\begin{array}{c} C_T \\ C_B \end{array} \right) \rightarrow \left(\begin{array}{c} C_0 \\ C_1 \\ C_2 \end{array} \right)$
where C_1 is a tile

$$\left(\begin{array}{c} A \\ C_1 \end{array} \right) := \text{QR TD} \left(\left(\begin{array}{c} A \\ C_1 \end{array} \right) \right)$$

Continue with
 $\left(\begin{array}{c} C_T \\ C_B \end{array} \right) \leftarrow \left(\begin{array}{c} C_0 \\ C_1 \\ C_2 \end{array} \right)$

endwhile

Fig. 2. Tiled algorithms for computing the QR factorization (continued)

2.2 Sequential Basic Building Kernels

Four basic building kernels (BK) appear highlighted in the previous algorithms: the QR factorization of a full dense matrix, the application of orthogonal transformations resulting from it, the QR factorization of a 2×1 blocked matrix with the top submatrix being upper triangular, and the application of the corresponding transformations to a 2×1 blocked matrix:

$$\begin{aligned} \text{BK1.} \quad A &:= \text{QR}(A), \\ \text{BK2.} \quad B &:= \text{APPLY_Q}(Q, B), \\ \text{BK3.} \quad \left(\begin{array}{c} R \\ C \end{array} \right) &:= \text{QR TD} \left(\left(\begin{array}{c} R \\ C \end{array} \right) \right), \quad \text{and} \\ \text{BK4.} \quad \left(\begin{array}{c} B \\ D \end{array} \right) &:= \text{APPLY_QTD} \left(Q, \left(\begin{array}{c} B \\ D \end{array} \right) \right), \end{aligned}$$

respectively. We note here that in the invocation of BK2 and BK4, Q is replaced by the matrix which contains the appropriate orthogonal transformations (stored in compact form in the strictly lower triangle).

The building kernels BK1 (QR) and BK2 (APPLY_Q) are well-known dense linear algebra operations, for which sequential efficient implementations exist as part of libflame and LAPACK legacy code (routines `geqrf` and `ormqr`) [16].

The key that makes the discussed tiled QR algorithm practical is the use of structure-aware implementations of BK3 (QR TD) and BK4 (APPLY_QTD) that

Algorithm:	$\begin{pmatrix} R \\ C \end{pmatrix} := \text{QR}_{\text{TD}} \left(\begin{pmatrix} R \\ C \end{pmatrix} \right)$
Partition	$R \rightarrow \begin{pmatrix} R_{TL} & R_{TR} \\ R_{BL} & R_{BR} \end{pmatrix}, C \rightarrow (C_L C_R)$
	where R_{TL} is 0×0 , C_L has 0 columns
while $m(R_{TL}) < m(R)$ do	
Determine block size b	
Repartition	
	$\begin{pmatrix} R_{TL} & R_{TR} \\ R_{BL} & R_{BR} \end{pmatrix} \rightarrow \begin{pmatrix} R_{00} & R_{01} & R_{02} \\ R_{10} & R_{11} & R_{12} \\ R_{20} & R_{21} & R_{22} \end{pmatrix}, (C_L C_R) \rightarrow (C_0 C_1 C_2)$
	where R_{11} is $b \times b$, C_1 has b columns
	$\begin{pmatrix} R_{11} \\ C_1 \end{pmatrix} := \text{QR} \left(\begin{pmatrix} R_{11} \\ C_1 \end{pmatrix} \right)$
	$\begin{pmatrix} R_{12} \\ C_2 \end{pmatrix} := \text{APPLY_Q} \left(\begin{pmatrix} R_{11} \\ C_1 \end{pmatrix}, \begin{pmatrix} R_{12} \\ C_2 \end{pmatrix} \right)$
Continue with	
	$\begin{pmatrix} R_{TL} & R_{TR} \\ R_{BL} & R_{BR} \end{pmatrix} \leftarrow \begin{pmatrix} R_{00} & R_{01} & R_{02} \\ R_{10} & R_{11} & R_{12} \\ R_{20} & R_{21} & R_{22} \end{pmatrix}, (C_L C_R) \leftarrow (C_0 C_1 C_2)$
endwhile	

Fig. 3. Blocked algorithm for the building block BK3

exploit the upper triangular form of the top submatrix. Figure 3 shows how to do so for BK3. Provided $b \ll t$, the procedure there requires $2t^3$ floating-point arithmetic operations (flops), which is considerably lower than the $8t^3/3$ flops required to compute the factorization if the structure of R is not considered and a general QR factorization of $\begin{pmatrix} R \\ C \end{pmatrix}$ was computed. The procedure for BK4 is shown in Figure 4, reducing the cost from $8t^3$ for the general procedure to $4t^3$ flops for the structure-aware one (provided $b \ll t$). For details, see [12].

2.3 Multi-threaded Basic Building Kernels

For multi-core processors, parallel implementations of the kernels BK1 and BK2 can be obtained by just linking the appropriate `libflame`/LAPACK routines with a multi-threaded implementation of BLAS. The result is a code that will extract all its parallelism from the invocations to BLAS from within the routines. Alternatively, one can also use multi-threaded implementations of `geqrf` and `ormqr` that are part of MKL for these two building kernels, or the parallel data-driven algorithm with dynamic scheduling described in [15].

The parallelization of the structure-aware building kernels BK3 and BK4 is more challenging. Of course, one can still link the sequential codes with a

Algorithm:	$\begin{pmatrix} B \\ D \end{pmatrix} := \text{APPLY_QTD} \left(\begin{pmatrix} R \\ C \end{pmatrix}, \begin{pmatrix} B \\ D \end{pmatrix} \right)$
Partition	$R \rightarrow \begin{pmatrix} R_{TL} & R_{TR} \\ R_{BL} & R_{BR} \end{pmatrix}, B \rightarrow \begin{pmatrix} B_T \\ B_B \end{pmatrix}, C \rightarrow (C_L C_R)$
	where R_{TL} is 0×0 , B_T has 0 rows, C_L has 0 columns,
while $m(R_{TL}) < m(R)$ do	
Determine block size b	
Repartition	
	$\begin{pmatrix} R_{TL} & R_{TR} \\ R_{BL} & R_{BR} \end{pmatrix} \rightarrow \begin{pmatrix} R_{00} & R_{01} & R_{02} \\ R_{10} & R_{11} & R_{12} \\ R_{20} & R_{21} & R_{22} \end{pmatrix}, \begin{pmatrix} B_T \\ B_B \end{pmatrix} \rightarrow \begin{pmatrix} B_0 \\ B_1 \\ B_2 \end{pmatrix},$
	$(C_L C_R) \rightarrow (C_0 C_1 C_2)$
	where R_{11} is $b \times b$, B_1 has b rows, C_1 has b columns
	$\begin{pmatrix} B_1 \\ D \end{pmatrix} := \text{APPLY_Q} \left(\begin{pmatrix} R_{11} \\ C_1 \end{pmatrix}, \begin{pmatrix} B_1 \\ D \end{pmatrix} \right)$
Continue with	
	$\begin{pmatrix} R_{TL} & R_{TR} \\ R_{BL} & R_{BR} \end{pmatrix} \leftarrow \begin{pmatrix} R_{00} & R_{01} & R_{02} \\ R_{10} & R_{11} & R_{12} \\ R_{20} & R_{21} & R_{22} \end{pmatrix}, \begin{pmatrix} B_T \\ B_B \end{pmatrix} \leftarrow \begin{pmatrix} B_0 \\ B_1 \\ B_2 \end{pmatrix},$
	$(C_L C_R) \leftarrow (C_0 C_1 C_2)$
endwhile	

Fig. 4. Blocked algorithm for the building block **bk4**

multi-threaded implementation of BLAS, yielding a parallel version of the codes that extracts all parallelism from within the calls to BLAS. However, given that b is small, low performance can be expected from this. We will refer to this first variant as *intra-tile column parallel*.

On the other hand, the application of transformations only occurs from the left, making the updates independent by slabs of columns. Thus, a highly parallel multi-thread implementation can be obtained by splitting the columns of the matrix that need to be updated into several slabs (as many as threads are being used) and then computing the update concurrently. We will refer to this second variant as *column parallel*.

3 OOC Algorithms for the QR Factorization

3.1 A Traditional OOC Implementation

Developers of OOC codes usually decide first how many tiles will be kept in-core, then carefully design their algorithms to reduce the number of data movements between main memory and disk, and finally insert the appropriate invocations

to I/O calls in the codes. In general, the tile size t is set to occupy a fraction of the main memory as large as possible: provided t is large enough, the I/O overhead necessary to move the tiles involved in a given operation is negligible compared with the cost of the computations ($O(t^2)$ accesses to disk vs. $O(t^3)$ flops). OOC algorithms that keep three to four tiles in-core are common for many dense linear algebra operations. To attain higher performance, computation and I/O have to be overlapped, and space is needed to store data in-core which will be involved in future computations (*double-buffering*). However, overlapping for performance also complicates programming, as asynchronous I/O routines need to be employed.

We explain next how to unburden the library developer from explicitly managing I/O and overlapping it with the computation.

3.2 A Run-Time System for OOC Dense Linear Algebra Operations

Our approach employs a run-time system which executes the codes corresponding to the algorithms for the tiled QR factorization (see Figures 11–14) in two stages. In the first stage, the run-time does a symbolic execution of the code creating a *list of pending tasks*: every time an invocation to a routine that corresponds to one of the building kernels is detected, a new annotation is introduced in the list which identifies the task to be computed and the operands (tiles) which are involved. Upon completion, tasks appear in this list in the same order as they are encountered in the codes.

During the second stage, the real I/O and computations occur. Here a *scout* thread and a *worker* thread collaborate to perform I/O and computations. The scout thread extracts the next task from the pending list, bringing in-core the tiles involved by the task. To hide memory latency, this thread uses a *software cache* with capacity to store a few tiles and the corresponding replacement policies/mechanisms. Once data is in-core for a given operation, the scout thread moves the task to the *list of ready tasks*, which only contains operations with all data in main memory.

The worker thread extracts the tasks from the list of ready tasks in order, one task at a time, and executes the corresponding operation using as many threads as cores are available in the system. Thus, in our current approach, parallelism is only exploited within the computations of a single task. However, in case there are two or more tasks in the ready list with all its input data updated (i.e., no previous task in the list will overwrite them), we could have also split the set of computational threads to execute them in parallel.

We have previously used the idea of a run-time/two-stage execution, with an initial symbolic analysis of the code, to extract more parallelism and improve the scalability in multi-core systems; see, e.g., [15]. The purpose is different here. In particular, we employ the run-time to overlap I/O done by the scout thread and computation performed by the worker thread without using asynchronous I/O routines. The fact that the list of tasks (or part of it) is known in advance, is equivalent in practice to having a perfect prefetch engine as the tiles that will be needed in future operations (tasks) are known *a priori*.

To ensure the correct operation of the worker thread, the replacement policy for the software cache only selects tiles which are not involved in any operation in the list of ready tasks. If there are no candidates which satisfy this criterion, the scout thread blocks till the execution of new tasks is completed.

The bottom line is that proceeding in this manner, the scout thread completely hides I/O from the library developer so that no changes are necessary in the in-core codes. Also, both threads conspire to hide asynchronous I/O from the developer.

The concurrent execution of the scout and the worker thread implies a first level of parallelism in the system. The use of multiple threads to perform the actual computations on the data reflects an additional, nested level of parallelism.

4 Experimental Results

The target architecture for the experiments is a workstation with two Intel Xeon QuadCore E5405 processors (8 cores) at 2.0 GHz with 8 GBytes of DDR2 RAM (peak performance is 128 billions of flops per second or GFLOPS). The Intel 5400 chipset provides an I/O interface with a peak bandwidth of 1.5 Gbits/second. The disk is a SATA-I with a total capacity of 160 Gbytes. All experiments were performed using MKL 10.0.1 and single precision.

We first evaluate the performance of the multi-threaded in-core building kernels, operating on matrices (tiles) of size $t \times t$ (BK1 and BK2) and $2t \times t$ (BK3 and BK4). Table 1 reports the number of invocations to each building kernel during the factorization of a square matrix of order n using the algorithms for the tiled (left-looking) QR factorization in Figures 1 and 2, with tile size t and $k = n/t$. There we also report the performance of the basic building kernels, measured in terms of GFLOPS, using the counts of $4t^3/3$, $2n^3$, $2t^3$ and $4t^3$ flops¹ for BK1, BK2, BK3 and BK4, respectively. In this analysis we set $t=5,120$ which, in a separate study, was found to be a fair value for the tiled OOC algorithm. Clearly, the performance of BK4 will determine the efficiency of the overall OOC algorithm and, therefore, we tried to optimize this building kernel carefully. In particular, the table shows the results of two parallelization strategies for BK4: one with all parallelism being extracted from calls to multi-threaded BLAS and an alternative one with a parallelization by blocks of columns (see subsection 2.3). In the experiments we also found that the block size $b=64$ (see Figures 3 and 4) was optimal in most cases.

Our second experiment evaluates the performance of several in-core and OOC routines for the QR factorization:

In-core MKL: The (in-core) multi-threaded implementation of the QR factorization in MKL 10.0.1.

In-core LAPACK: The (in-core) LAPACK legacy code linked with the multi-threaded BLAS in MKL 10.0.1.

¹ We emphasize that we are only counting “useful computations” and do not count additional operations that are artificially introduced in order to expose better parallelism and/or improve the use of matrix-multiplications.

Table 1. Number of invocations and performance of different implementations of the multi-threaded in-core building kernels operating on tiles of size $t=5,120$

Building kernel	#invocations	Performance	
		GFLOPS	Parallelization strategy
BK1	k	45	Multi-threaded BLAS
BK2	$k^3/2$	67	Multi-threaded BLAS
BK3	$k^2/2$	39	Multi-threaded BLAS
BK4	$k^3/3$	53	Multi-threaded BLAS
		65	Column parallel

OOC explicit+intra-tile column parallel: OOC implementation with explicit invocations to I/O routines (see subsection 3.1). The tile size was set in this routine to $t=5,120$. Although, in theory, using a large tile size makes the cost of moving data between disk and main memory ($O(t^2)$ disk accesses) negligible compared with the computational cost ($O(t^3)$ flops), in our experimentation we found out a large drop in the disk transfer rate for tiles of dimension larger than $5,120 \times 5,120$. (We experienced similar behaviour for several other current desktop systems equipped with different disks.) In this routine, parallelism is extracted implicitly by linking to a multi-threaded BLAS (see subsection 2.3).

OOC cache+column parallel: OOC implementation with a software cache in place to reduce the number of I/O transfers (see Section 3.2). The cache occupies 6 GBytes in RAM, and is organized as a k -way set associative with $t=5,120$ and $k = n/t$. LRU was implemented as the replacement policy. Parallelism is extracted explicitly by implementing column parallel variants for the building kernels BK3 and BK4 (see subsection 2.3). I/O is synchronous.

OOC cache+I/O overlap+column parallel: OOC implementation that includes all mechanisms of the previous routine plus overlap of I/O and computation (see Section 3.2).

Figure 5 reports the performance of these routines measured in terms of GFLOPS, with the usual count of $4n^3/3$ flops for the QR factorization of a square matrix of order n . (Experiments with nonsquare matrices offered similar results. We also note here that the tiled QR factorization performs a larger number of flops than the in-core factorization routines in MKL and LAPACK; see [2] for details.)

The results in the figure show a practical peak performance for the in-core QR factorization that is slightly over 74 GFLOPS. Using routine **OOC column parallel+cache+I/O overlap**, the tiled QR factorization which operates on OOC data yields a performance that is around 65 GFLOPS and, therefore, close to that of the in-core algorithm. As expected, the performance of the tiled OOC algorithm matches that of the building kernel BK4, in a practical demonstration that the disk latency is mostly hidden. The performance results reveal that the GFLOPS rate for the OOC algorithm is maintained as the problem size is increased, thus confirming the scalability of the solution.

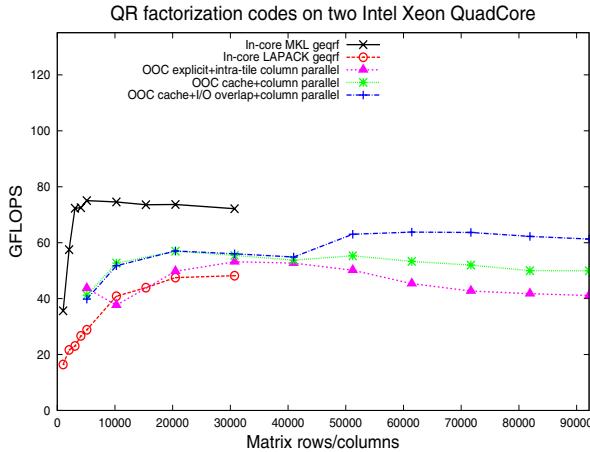


Fig. 5. Performance of the QR factorization codes on a multi-core processor

To asses the benefits contributed by the use of the software cache, Table 2 shows the number of tiles read from or written to disk for the OOC routine with explicit I/O calls and the ones that employ a software cache. The results demonstrate that the software cache greatly reduces the number of tiles that are transferred between the disk and RAM.

Table 3 reports the execution time required to compute the tiled QR factorization for routine **OOC column parallel+cache+I/O overlap** as well as the amount of memory that is needed to store the full dense matrix. The results show that what would have been considered a very large problem only a few years ago was solved in less than 5 hours. Solving a linear least squares problem or a linear system (with a few right-hand sides) once the orthogonal matrix and the triangular factor are available is computationally much less expensive

Table 2. Reduction in the number of disk accesses (in terms of number of tiles read/s/written) attained by the use of a software cache

Matrix size (square)	OOC explicit		OOC cache	
	#reads	#writes	#reads	#writes
51,200	1,045	715	237	100
92,160	5,985	4,047	2,170	324

Table 3. Execution time (in hours, minutes, and seconds) of the tiled QR factorization code using the OOC run-time and amount of memory needed to hold the matrix

Matrix size (square)	Time		MBytes
	54min	20.9sec	
51,200			10,000
92,160	4h 43min	12.3sec	40,000

than the factorization procedure and does not represent a challenge from the viewpoint of an OOC implementation.

5 Concluding Remarks

In this paper we have described and evaluated a run-time system which deals with I/O, overlaps computation and data movement from disk, and implements a software cache and the associated mechanisms. The case study is the QR factorization, a complex operation for which tiled in-core left-looking codes exist. The results obtained by combining these codes with the run-time system show that the extension of the memory hierarchy to include the disk can be made transparent to the library developer at the expense of little overhead. Our approach thus greatly increases the programmer's productivity without significantly hurting performance.

This work also demonstrates that multi-core processors are a cost-effective approach for the solution of many dense linear algebra operations of moderate scale. The implicit message is that, for these problems, it is not necessary to utilize expensive distributed-memory architectures and design complex message-passing algorithms, provided one is willing to wait longer.

Acknowledgements

The researchers at the Universidad Jaime I were supported by projects CICYT TIN2005-09037-C02-02, TIN2008-06570-C04-01 and FEDER, and P1B-2007-19 of the *Fundación Caixa-Castellón/Bancaixa* and UJI.

References

1. Baboulin, M., Giraud, L., Gratton, S., Langou, J.: Parallel tools for solving incremental dense least squares problems. application to space geodesy. Technical Report UT-CS-06-582; TR/PA/06/63, University of Tennessee; CERFACS (2006); To appear in J. of Algorithms and Computational Technology 3(1) (2009)
2. D'Azevedo, E.F., Dongarra, J.J.: The design and implementation of the parallel out-of-core scalapack LU, QR, and Cholesky factorization routines. LAPACK Working Note 118 CS-97-247, University of Tennessee, Knoxville (1997)
3. Reiley, W.C., van de Geijn, R.A.: POOLAPACK: Parallel Out-of-Core Linear Algebra Package. Technical Report CS-TR-99-33, Department of Computer Sciences, The University of Texas at Austin (1999)
4. Toledo, S.: A survey of out-of-core algorithms in numerical linear algebra. In: DIMACS Series in Discrete Mathematics and Theoretical Computer Science (1999)
5. Marqués, M., Quintana-Ortí, G., Quintana-Ortí, E.S., van de Geijn, R.: Solving “large” dense matrix problems on multi-core processors. In: 10th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing – PDSEC 2009 (to appear, 2009)
6. Van Zee, F.G.: The complete reference (2008) (in preparation),
<http://www.cs.utexas.edu/users/flame>

7. Baboulin, M.: Solving large dense linear least squares problems on parallel distributed computers. Application to the Earth's gravity field computation. Ph.D. dissertation, INPT, TH/PA/06/22 (2006)
8. Gunter, B.C.: Computational methods and processing strategies for estimating Earth's gravity field. PhD thesis, The University of Texas at Austin (2004)
9. Geng, P., Oden, J.T., van de Geijn, R.: Massively parallel computation for acoustical scattering problems using boundary element methods. *Journal of Sound and Vibration* 191(1), 145–165 (1996)
10. Schafer, N., Serban, R., Negrut, D.: Implicit integration in molecular dynamics simulation. In: ASME International Mechanical Engineering Congress & Exposition (2008) (IMECE2008-66438)
11. Zhang, Y., Sarkar, T.K., van de Geijn, R.A., Taylor, M.C.: Parallel MoM using higher order basis function and PLAPACK in-core and out-of-core solvers for challenging EM simulations. In: IEEE AP-S & USNC/URSI Symposium (2008)
12. Gunter, B.C., van de Geijn, R.A.: Parallel out-of-core computation and updating the QR factorization. *ACM Transactions on Mathematical Software* 31(1), 60–78 (2005)
13. Watkins, D.S.: *Fundamentals of Matrix Computations*, 2nd edn. John Wiley & Sons, Inc., New York (2002)
14. Dongarra, J.J., Du Croz, J., Hammarling, S., Duff, I.: A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software* 16(1), 1–17 (1990)
15. Quintana-Ortí, G., Quintana-Ortí, E.S., van de Geijn, R., Zee, F.V., Chan, E.: Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Transactions on Mathematical Software* (2008) (to appear), FLAME Working Note #32, <http://www.cs.utexas.edu/users/flame/>
16. Anderson, E., Bai, Z., Demmel, J., Dongarra, J.E., DuCroz, J., Greenbaum, A., Hammarling, S., McKenney, A.E., Ostrouchov, S., Sorensen, D.: *LAPACK Users' Guide*. SIAM, Philadelphia (1992)

Adaptive Parallel Householder Bidiagonalization*

Fangbin Liu¹ and Frank J. Steinstra²

¹ ISLA, Informatics Institute, University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
fliu@science.uva.nl

² Department of Computer Science, VU University
De Boelelaan 1081A, 1081 HV Amsterdam, The Netherlands
fjseins@cs.vu.nl

Abstract. With the increasing use of large image and video archives and high-resolution multimedia data streams in many of today's research and application areas, there is a growing need for multimedia-oriented high-performance computing. As a consequence, a need for algorithms, methodologies, and tools that can serve as support in the (automatic) parallelization of multimedia applications is rapidly emerging.

This paper discusses the parallelization of Householder bidiagonalization, a matrix factorization method which is an integral part of full Singular Value Decomposition (SVD) — an important algorithm for many multimedia problems. Householder bidiagonalization is hard to parallelize efficiently because the total number of matrix elements taking part in the calculations reduces during runtime. To overcome the growing negative performance impact of load imbalances and overprovisioning of compute resources, we apply adaptive runtime techniques of *periodic matrix remapping* and *process reduction* for improved performance. Results show that our adaptive parallel execution approach provides a significant improvement in efficiency, even when applying a set of compute resources which is (initially) very large.

1 Introduction

The research area of Multimedia Content Analysis (MMCA) considers all aspects of the automatic extraction of knowledge from large multimedia archives and video streams. The MMCA domain is rapidly facing a computational problem of phenomenal proportions, as digital video produces high data rates, and multimedia archives steadily run into Petabytes of storage. As individual desktop computers can no longer satisfy the increasing computational demands, the use of compute clusters, grids, and cloud systems is rapidly becoming indispensable.

A common way to help researchers and developers in the MMCA domain in the development of high-performance applications is to provide a library of

* This research is supported by the Netherlands Organization for Scientific Research (NWO) under NWO-GLANCE grant no. 643.000.602.

pre-parallelized building block operations that hide the complexities of parallelization behind a sequential programming interface. Over the past years we have developed one such library for *user transparent* parallel multimedia computing on compute clusters, called Parallel-Horus, which has been applied for implementation of a number of state-of-the-art MMCA applications [1].

While Parallel-Horus incorporates most algorithms commonly applied in the field, it is essential to expand the set of operations to further enhance the library's applicability. Important missing functionality is that of matrix factorization by way of Singular Value Decomposition (SVD). More specifically, in this paper we focus on the most interesting — and most computationally demanding — part of one approach to SVD factorization, i.e.: Householder bidiagonalization.

Parallel solutions to Singular Value Decomposition in general, and Householder bidiagonalization in particular, have been studied extensively in the literature [2], [3], [4], [5], [6]. In contrast to such earlier efforts, our main focus is on the integration of a parallel solution to Householder bidiagonalization behind a user transparent parallel programming interface. As such, the foremost research question underlying the work described in this paper is stated as follows: can we implement a parallel Householder bidiagonalization method based on the basic building block operations available in Parallel-Horus, such that it integrates effortlessly, yet efficiently, into our user transparent parallel library?

Householder bidiagonalization is hard to parallelize efficiently. This is because the size of the *working set* (i.e., the number of matrix elements taking part in the calculations) reduces over time. As a result, while it is often beneficial to use a large number of compute nodes in the early stages of the execution, the cost of parallelization and load imbalances eventually outweigh the cost of actual calculations, to the effect that obtained speedups (if at all) are generally low.

In the solution proposed in this paper we apply an approach called *periodic matrix remapping* for load balancing. Further, to optimize performance under the continuous reduction of the working set, we apply an approach called *process reduction* to gradually reduce the number of compute nodes at runtime. The decision making for process reduction is based on a performance model that continuously compares performance results obtained on the current set of nodes with estimations for a reduced number of nodes. Extensive evaluation of our solution to parallel Householder bidiagonalization shows that we obtain high speedups, even for a large number of compute nodes.

This paper is organized as follows. Section 2 introduces the Parallel-Horus multimedia computing library. Section 3 introduces the Householder bidiagonalization method. Section 4 discusses our adaptive parallel Householder bidiagonalization approach, and presents a simple performance model used for performance optimization. Subsequently, Section 5 gives an evaluation of the obtained performance and speedup results. Concluding remarks are given in Section 6.

2 Parallel-Horus

Parallel-Horus is a cluster programming framework that allows programmers to implement data parallel multimedia applications as fully sequential programs.

The Parallel-Horus framework consists of commonly used multimedia data types and associated operations, implemented in C++ and MPI. The library's API is made identical to that of an existing sequential library: Horus [7]. Similar to other frameworks [8], Horus recognizes that a small set of *algorithmic patterns* can be identified that covers the bulk of all multimedia-oriented functionality.

Horus includes patterns for all such functionality, including unary and binary pixel operations, global reduction, generalized convolution, iterative and recursive neighborhood operations, and geometric transformations. Current developments include patterns for operations on large datasets, as well as patterns on increasingly important data structures, such as feature vectors obtained from earlier calculations. For reasons of efficiency, all Parallel-Horus operations are capable of adapting to the performance characteristics of a parallel machine at hand, i.e. by being flexible in the partitioning of data structures. Moreover, it was realized that it is not sufficient to consider parallelization of library operations *in isolation*. Therefore, the library was extended with a run-time approach for communication minimization (called *lazy parallelization*) that automatically parallelizes a fully sequential program at runtime by inserting communication primitives and additional memory management operations whenever necessary.

Results for realistic multimedia applications have shown the feasibility of the Parallel-Horus approach, with data parallel performance (obtained on individual cluster systems) consistently being found to be optimal with respect to the abstraction level of message passing programs [1]. Notably, Parallel-Horus was applied in recent NIST TRECVID benchmark evaluations for content-based video retrieval, playing a crucial role in achieving top-ranking results in a field of strong international competitors [9], [10]. Moreover, recent extensions to Parallel-Horus, that allow for services-based multimedia Grid computing, have been applied successfully in large-scale distributed systems, involving hundreds of massively communicating compute resources covering our entire globe [1]. Real-time and off-line applications implemented with this extended system have resulted in a 'best technical demo award' at ACM Multimedia 2005 [11] and a 'most visionary research award' at AAAI 2007 [12]. Also, Parallel-Horus has been used in our prize-winning contribution to the First IEEE International Scalable Computing Challenge at CCGrid 2008 (Lyon, France) [13].

Clearly, Parallel-Horus is a system that serves well in bringing the benefits of high-performance computing to the multimedia community, but we are constantly working on further improvements, as exemplified in the following sections. Prototypical code (in C and MPI) of an earlier proof-of-concept implementation of Parallel-Horus, as well as of several example image processing applications, is available at: <http://www.science.uva.nl/~fjseins/ParHorusCode/>.

3 Singular Value Decomposition

In linear algebra, Singular Value Decomposition (SVD) is an important factorization method for rectangular real or complex matrices. The method is used in many application areas, including a.o. numerical weather prediction and

multimedia content analysis. As stated in [14], a real $m \times n$ matrix A with $m \geq n$ can be decomposed into three matrices:

$$A = U\Sigma V^T, U : m \times n, V : n \times n \quad (1)$$

where $U^T U = V^T V = VV^T = I_n$, matrix $\Sigma = \text{diag}(\sigma_1, \dots, \sigma_n)$, and diagonal elements $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_n \geq 0$ being the *singular values* of matrix Σ .

As shown in [15], the SVD factorization for a matrix A can be performed in two steps. First, matrix A is reduced into *upper bidiagonal form* by way of a series of *Householder transformations*. Second, the QR algorithm is performed to find the singular values of the upper bidiagonal matrix. These two phases combined properly produce the singular value decomposition of matrix A . In this paper, we will focus only on the first phase of the SVD factorization (i.e., the Householder bidiagonalization), as it is the most computationally demanding part of the calculation.

3.1 Householder Bidiagonalization

Householder bidiagonalization, or the reduction of input matrix A into upper bidiagonal form, proceeds by alternately pre- and post-multiplying A by so-called *Householder transformations*

$$P^{(k)} = I - 2x^{(k)}x^{(k)T}, \text{ with } k = 1, 2, \dots, n$$

and

$$Q^{(k)} = I - 2y^{(k)}y^{(k)T}, \text{ with } k = 1, 2, \dots, n-2$$

where $x^{(k)T}x^{(k)} = y^{(k)T}y^{(k)} = 1$, such that

$$P^{(n)} \dots P^{(1)} A Q^{(1)} \dots Q^{(n-2)} = \begin{Bmatrix} q_1 & e_2 & 0 & \dots & 0 \\ 0 & q_2 & e_3 & & \vdots \\ \vdots & & \ddots & \ddots & 0 \\ \vdots & & & \ddots & e_n \\ 0 & \dots & \dots & 0 & q_n \\ \vdots & & & & \vdots \\ 0 & \dots & \dots & \dots & 0 \end{Bmatrix} \quad (2)$$

Before discussing the parallelization of this Householder bidiagonalization in the next Section, we will first present a thorough overview of the data dependencies involved in the underlying algorithm.

3.2 Data Dependencies

Generally speaking, Householder bidiagonalization takes place in n iterations, where each iteration involves one pre-multiplication and one post-multiplication

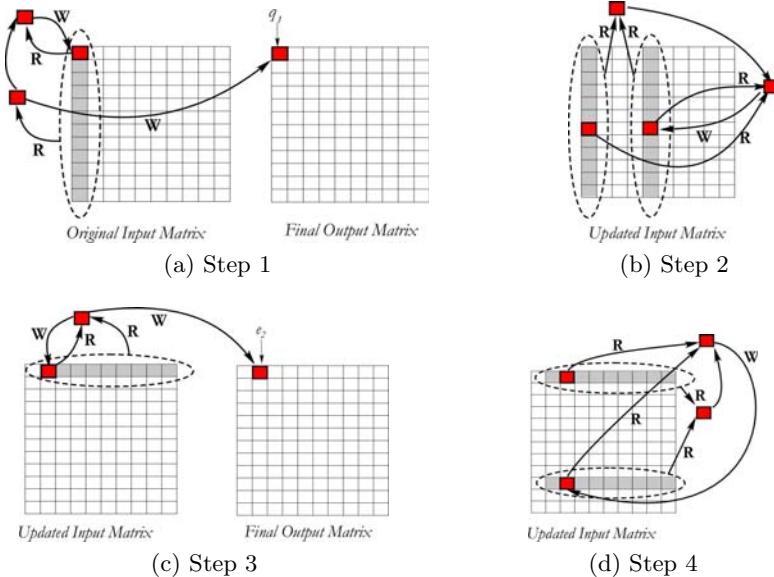


Fig. 1. Four steps in a single Householder iteration. R = read access; W = write access

resulting in a new (intermediate) $m \times n$ matrix $H^{(i)}$. In each iteration i (with $0 \leq i \leq n$) all elements in column i of $H^{(i)}$ are being set to 0, except for diagonal element q_i and the upper diagonal element e_i directly above it (if it exists). Similarly, all elements in row i of $H^{(i)}$ are being set to 0 except for diagonal element q_i and the upper diagonal element e_{i+1} directly to its right (if it exists). Furthermore, all elements in the right-lower $(m-i) \times (n-i)$ submatrix of $H^{(i)}$ are being updated. In iteration $i+1$ this right-lower $(m-i) \times (n-i)$ submatrix of $H^{(i)}$ is being used as the basis for all calculations. The set of all elements in this $(m-i) \times (n-i)$ submatrix is what we refer to as the *working set* for iteration $i+1$.

In practice, a single iteration of Householder bidiagonalization consists of four major consecutive steps (see Figure 1). While all i iterations are very similar, for ease of presentation we focus on the first iteration only (i.e.: $i = 1$). In the first of the four steps, the first diagonal element q_1 is calculated and stored in the final bidiagonal output matrix. As shown in Figure 1(a), the calculation of q_1 is dependent on all values in column 1. In addition, a new (temporary) value for $A[1, 1]$ is calculated and stored in-place, on the basis of q_1 and $A[1, 1]$ itself.

In the second step, all elements in all columns to the right of column 1 are being updated. As shown in Figure 1(b), each updated element $A[x, y]$ is dependent on all elements in column 1 as well as on all elements in column y . The update of each matrix element in this second step is again in-place.

Step 3 is rather similar to Step 1. In this step, the first upper diagonal element e_2 is calculated and stored in the final bidiagonal output matrix. As shown in Figure 1(c), the calculation of e_2 depends on all values in row 1, except the first.

As in the first step, a new (temporary) value for $A[1, 2]$ is calculated and stored in-place, on the basis of e_2 and $A[1, 2]$.

The fourth step is very similar to Step 2. In this final step, all elements in all rows below row 1 are being updated, except for the first element in each row. Figure 1(d) shows that each updated element $A[x, y]$ is dependent on all elements in row 1 as well as on all elements in row x . As before, the update of each matrix element is an in-place operation.

The completion of these four calculation steps also completes one iteration of the Householder bidiagonalization. As stated above, the right-lower sub-diagonal matrix of the matrix that results after Step 4 constitutes the working set for the next iteration. In practice this means that each subsequent iteration uses an smaller matrix as input: i.e. the output matrix of Step 4 excluding its first row and its first column. As discussed in the following Section, this continuous reduction of the working set has important consequences for parallel execution.

4 Parallelization

In the parallelization of the Householder bidiagonalization, two strategic choices must be made: the parallel execution of a single Householder iteration, and the parallel execution of all iterations in turn. Below, we will first present our considerations with respect to the parallelization of a single iteration. This is followed by a discussion of the parallel execution of the full Householder bidiagonalization.

4.1 Parallelizing a Single Iteration

In this section we focus on the parallelization of a single iteration of the algorithm as presented in Figure 1, as well as on the integration with the Parallel-Horus library. For reasons beyond the scope of this paper, Parallel-Horus only offers implementations on the basis of the paradigm of *data parallelism*. As a consequence, we restrict our considerations to data parallel solutions only.

Parallel-Horus offers a rich set of data partitioning routines, which allow data structures to be split up in a multitude of ways. The most commonly used data partitioning routines provide splitting and scattering of data structures in a horizontal (or row-wise), vertical (or column-wise), and hybrid (or block-wise) manner. Each of these data partitioning routines ensures that the workload is spread in a balanced way: each compute node in the system will get an (approximately) equal subsection of the entire data structure.

In principle, each of these three data partitioning strategies is a good candidate for parallelization of a single iteration of the Householder bidiagonalization. When considering the data dependencies discussed in the previous section, however, there are several issues that are of importance. In case of column-wise (vertical) partitioning of the data structures involved in the Householder algorithm, we see that Step 1 can be executed without problems. In fact, only one of the nodes has work to do — and it can do so immediately because all data dependencies are resolved locally (i.e. without the need for inter-node communication). Step 2 is only marginally more problematic. The updating of the matrix

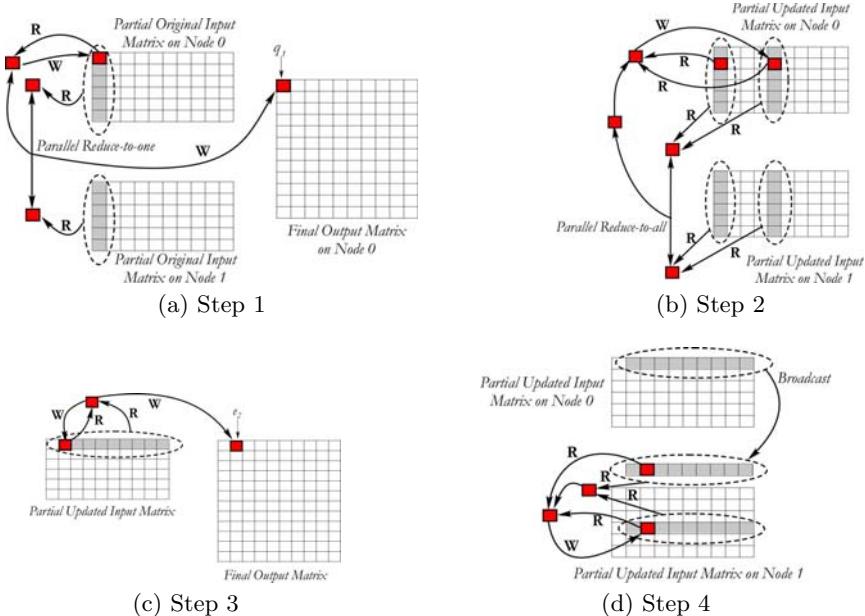


Fig. 2. A single parallel Householder iteration by way of horizontal data partitioning

elements in this step requires each node to have access to the first matrix column. Hence, this column would have to be provided (broadcast) to all compute nodes in the system. Steps 3 and 4 are more complicated, as in both steps many of the data dependencies are not local. As a result, inter-node communication (by way of parallel reduction) would be needed to resolve these dependencies.

Clearly, a similar line of reasoning holds for row-wise partitioning. Cyclic block-wise partitioning, on the other hand, would cause non-local data dependencies to exist in all four steps of the algorithm, so we do not consider this approach any further. One further approach, which would overcome much of the non-local data dependencies as existent in both vertical and horizontal partitioning, is to use vertical partitioning in the first two steps, and horizontal partitioning in the last two. This, however, would require a complete matrix re-distribution — causing significant communication overhead — in the heart of the parallel algorithm. Initial experiments with this parallelization approach applied to various numbers of nodes in a real cluster system indeed showed an increase in the execution time of a factor of two or more, when compared to the approach based on horizontal partitioning. As a result, we also do not consider this two-step vertical-horizontal partitioning approach further.

Finally, a comparison of results obtained for horizontal versus vertical partitioning — which theoretically should give identical results — showed that horizontal partitioning is always significantly faster (on average about 20%). This is explained by the fact that, in the case of horizontal partitioning, most of the data structures to be communicated are stored contiguously in memory. In contrast,

vertical partitioning causes data structures to be stored non-contiguously, a problem which is discussed extensively in [16]. As a result, horizontal partitioning is our parallelization strategy of choice for each single iteration of the Householder bidiagonalization (see Figure 2).

4.2 Executing Multiple Iterations

In this section we build further on the parallelization approach presented above, and discuss the parallelization of consecutive Householder iterations. As stated in Section 3, one important problem of our algorithm is that the working set (i.e.: the total number of matrix elements taking part in the calculations) reduces over time. Specifically, in comparison to the matrix used in iteration i , the matrix used in iteration $i + 1$ has lost one row and one column.

The continuous reduction of the size of the working set has two important consequences for our parallelization strategy. First, if we would apply the original horizontal partitioning for all iterations without change, the parallel execution would face a gradually increasing *load imbalance*. This is because the loss of matrix elements is not evenly distributed over all compute nodes. The node that has been provided with the uppermost part of the matrix will be the first to gradually loose all its data *before* any of the other nodes loses even a single matrix row (note that, at the same time, all partial matrices on all nodes do loose columns in an evenly distributed manner). When the algorithm would progress in this way, eventually only a single node would be calculating on the remaining data — with all other nodes wasting idle cycles.

A second problem appears when we assume that we would be able to implement a mechanism that overcomes load imbalances. With such a mechanism the algorithm would run at a maximum level of parallelism: all nodes would be kept busy until the final phase of the calculations. While generally beneficial, such a mechanism would still not ensure maximum performance. This is because the amount of calculations to be performed in each iteration decreases at a faster rate than the amount of communication required to resolve all data dependencies. In other words, the computation versus communication ratio changes over time: in the initial stages of the calculations this ratio is generally high, but it gradually decreases while the algorithm progresses. Eventually, the time spent on communication can (en will) overtake the time spent on actual calculations. Stated differently, the parallel algorithm will suffer from a gradual *overprovisioning of compute resources*.

Adaptive Parallelization

To overcome the first problem of load imbalances we apply a method called *periodic matrix remapping*. In general, this means that for a parallel system consisting of n nodes the working set is repartitioned after each n iterations of the Householder algorithm. After the repartitioning has taken place, the working set is again evenly distributed over all nodes in the system. In practice, we have implemented the repartitioning by way of *upward matrix row-shifting*, meaning

that each compute node communicates a number of rows to its upper neighbor and receives a (smaller) number of rows from its lower neighbor.

To overcome the additional problem of the overprovisioning of resources, we have extended our periodic matrix remapping approach with an approach called *process reduction*. Essentially, this approach causes a gradual decrease in the number of compute nodes applied in the calculations, to the effect that the final phase of the algorithm may (and generally will) use just a single compute node — with all remaining nodes being available for other tasks.

Clearly, for optimal performance one needs to carefully select the moment and frequency at which process reductions take place. Reducing the number of nodes too early leads to a (temporary) underprovisioning of compute resources. Doing it too late or too infrequently may cause the cost of the communication performed by the algorithm to outweigh the cost of computation for too long. Reducing the number of nodes too often also will downgrade the performance of the algorithm, due to the inherent communication needed for the process reductions themselves.

Performance Estimation

For performance optimization we apply a performance model that can estimate the parallel performance for a single iteration of the Householder algorithm on any given number of nodes. The results of the model are used to steer the decision making for the execution of process reduction.

The applied performance model is based on our earlier work on the estimation of the computation costs and the communication costs of data parallel image and video applications [16], [17]. A full discussion of the underlying modeling techniques is far beyond the scope of this paper. At a high level of abstraction, however, we can state that the execution time T_{seqHH} of the sequential computations executed in a single Householder iteration performed on a $m \times n$ input matrix can be approximated by

$$T_{seqHH} = (2m - 1) \times (n - 1) \times C_{update}$$

where C_{update} is the cost of updating a single matrix element, as performed in Steps 2 and 4 of our sequential algorithm (see Figure 1). The (static) benchmarking process for obtaining an accurate value for this model parameter is explained in [17]. Note that we have abstracted away all other parts of the sequential calculations, which is acceptable for our purposes.

The execution time T_{parHH} for parallel execution of the computations taking place in our algorithm is simply obtained by

$$T_{parHH} = \frac{T_{seqHH}}{nrCPUs}.$$

In addition, the execution time T_{comm} of the communication steps executed in a single Householder iteration can be approximated by

$$T_{comm} = (2n - 1) \times C_{bcast}$$

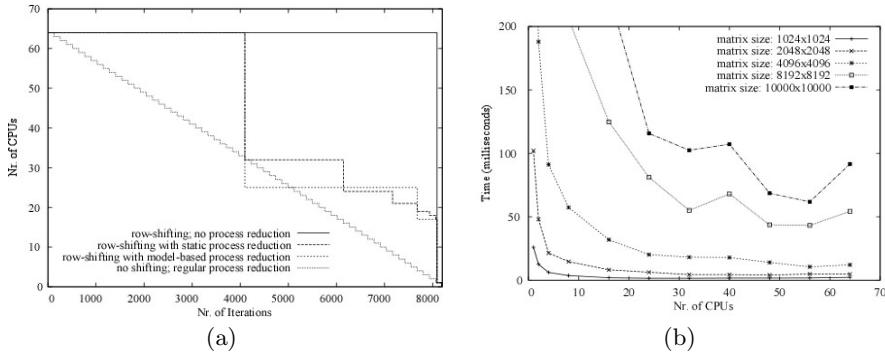


Fig. 3. (a) Four adaptive parallel execution strategies; (b) execution times for a single Householder iteration using matrices ranging from 1024^2 to 10000^2

where C_{bcast} is the cost for performing a broadcast of a 8-byte double value in a parallel system consisting of $nrCPUs$ nodes. A description of the modeling of such a broadcast operation, and the required benchmarking process, is given in [16]. Note that, again, we have abstracted away several communication steps, including a parallel reduce-to-one operation.

We have evaluated our model by comparing measurement results obtained for a single Householder iteration with our model predictions. At all times, our predictions were within 3 to 5% of the measurements, a result which is entirely in line with our earlier results presented in [16], [17].

5 Evaluation

We have tested our implementations on one of the clusters part of the 5-cluster Distributed ASCI Supercomputer 3 (DAS-3) installed in The Netherlands. The cluster, located at VU University (Amsterdam), consists of 85 dual-CPU/dual-core 2.4 GHz AMD Opteron DP 280 compute nodes (each having 4 GB of memory), all of which are linked via a high speed Myri-10G interconnect.

In our evaluation we present results for five different parallel execution strategies. The first strategy only applies horizontal data partitioning, but does not apply adaptive parallel execution. Essentially, this strategy serves as the basis for our comparison. In the second strategy, we apply upward matrix row-shifting for load balancing, but we do not apply process reduction. In the third approach, we perform process reduction at regular intervals (i.e.: whenever a node is found to have an empty working set), without performing load balancing. In essence, these latter two approaches constitute two extreme ends in the range of adaptive parallel execution strategies. Our two remaining strategies do apply load balancing by way of row-shifting as well as process reduction. The two strategies differ in that the first applies process reductions at fixed (pre-selected) instances, while the second approach makes all decisions regarding process reductions on the basis of our performance model. Figure 3(a) presents the gradual reductions taking place for each of the four *adaptive* strategies.

#CPUs	RS: yes PR: model	RS: yes PR: no	RS: no PR: no	RS: no PR: regular	RS: yes PR: static
1	3495.16	3495.16	3495.16	3495.16	3495.16
2	2061.20	2106.58	2627.18	2648.88	2664.30
4	984.36	979.17	1323.32	1324.44	1256.69
8	474.32	473.26	846.65	662.22	637.54
16	304.15	309.04	346.42	348.23	439.78
24	177.17	217.15	241.20	247.67	294.20
32	144.65	173.28	188.91	200.24	247.87
40	125.55	152.38	163.45	177.42	220.75
48	117.83	132.02	160.52	163.11	195.21
56	113.50	133.71	156.80	160.98	192.43
64	112.84	131.48	150.13	156.11	189.43

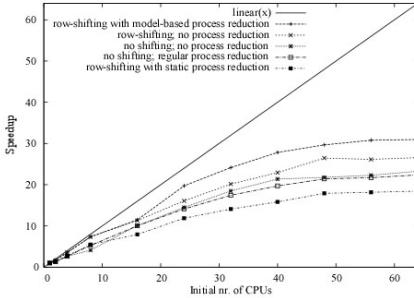


Fig. 4. Performance (left) and speedup (right) results for our five parallel execution strategies using a matrix of 8192^2 . RS = row-shifting; PR = process reduction.

Before presenting results for complete runs of the algorithm, we will first briefly evaluate the data parallel performance of a single Householder iteration. We have measured the execution time for a single iteration applied to a wide range of input matrix sizes on a varying number of nodes. As expected, and as shown in Figure 3(b), for fastest execution the number of nodes to be used is dependent on the size of the input matrix: for smaller matrices it is beneficial to use a smaller number of compute nodes. From this we conclude that, with a gradually decreasing working set, process reduction should be beneficial indeed.

Performance in seconds and speedup characteristics for our five parallel execution strategies are given in Figure 4. As can be seen, our implementation based on performance modeling — applied to a matrix of size 8192^2 — provides highest performance. All other strategies lag behind, with the implementation based on static (fixed) process reduction being the slowest. Interestingly, the non-adaptive parallel implementation still is the third fastest — even though it is significantly slower than our model-based approach. From this, we conclude that the combination of periodic matrix remapping and process reduction indeed can give improved performance, but only if the decision making process underlying these approaches cleverly incorporates the runtime characteristics of the algorithm on the specific parallel machine at hand. When periodic matrix remapping and process reduction are applied without care, the extra communication steps induced by these optimization strategies may prove too expensive.

With respect to the speedup graph of Figure 4 we need to state that the three graphs for runs including process reduction are given for the *initial* number of compute nodes, without identifying subsequent process reductions. This means that one can not simply calculate efficiency figures by dividing the obtained speedup by the indicated number of CPUs. Such a figure would merely provide a lower bound on the obtained efficiency. One way to solve this problem is to normalize our results for the actual number of nodes used in each execution phase. Our results show that we obtain a normalized efficiency for a 64-node run of 52.4% for our model-based solution, while the non-adaptive execution strategy obtains only 36.4%. Hence, our model-based approach provides an efficiency improvement of more than 44%. Moreover, the power of our model-based approach

is shown by the fact that, for a smaller number of initial nodes (in this case up to 24 nodes), close-to-linear speedup is obtained. Our results for different matrix sizes, and for different sizes of the parallel system, give a similar picture.

6 Conclusions

In this paper we have presented an adaptive parallel execution strategy for Householder bidiagonalization — an important algorithm in (a.o.) multimedia content analysis. The algorithm was implemented by using (as much as possible) the basic building blocks for data parallel processing available in the Parallel-Horus parallel multimedia computing library.

The Householder algorithm is hard to parallelize efficiently, as it suffers from a gradual decrease in the number of matrix elements used in the calculations. To overcome the negative performance impact of load imbalances and overprovisioning of compute resources, we have applied adaptive runtime techniques of *periodic matrix remapping* and *process reduction* for improved performance. Our results show that the combination of periodic matrix remapping and process reduction can improve performance, but only if the runtime characteristics of the algorithm are taken into account. We have shown that our model-based adaptive parallel execution approach can improve the obtained efficiency of a non-adaptive execution strategy by 44% or more. Moreover, in contrast to other strategies, our model-based approach obtains close-to-linear speedups when excessive overprovisioning of initial compute nodes is avoided.

References

1. Seinstra, F., et al.: High-Performance Distributed Video Content Analysis with Parallel-Horus. *IEEE Multimedia* 14(4), 64–75 (2007)
2. Chu, E., George, J.: Gaussian Elimination with Partial Pivoting and Load Balancing on a Multiprocessor. *Parallel Computing* 5(1), 65–74 (1987)
3. Dongarra, J., Sorensen, D.: A Fully Parallel Algorithm for the Symmetric Eigenvalue Problem. *SIAM J. Sci. Stat. Comput.* 8(2), 139–154 (1987)
4. Egecioglu, O., et al.: Givens and Householder Reductions for Linear Least Squares on a Cluster of Workstations. In: Proc. HiPC 1995, December 1995, pp. 734–739 (1995)
5. Groser, B., Lang, B.: Efficient Parallel Reduction to Bidiagonal Form. *Parallel Computing* 25(8), 969–986 (1999)
6. Rabani, E., Toledo, S.: Out-of-core SVD and QR Decompositions. In: Proceedings of PPSC 2001, Portsmouth, USA (March 2001)
7. Koelma, D., et al.: Horus C++ Reference, v. 1.1. Technical report, University of Amsterdam, The Netherlands (January 2002)
8. Morrow, P., et al.: Efficient Implementation of a Portable Parallel Programming Model for Image Processing. *Concur. Pract. Exp.* 11, 671–685 (1999)
9. Seinstra, F., et al.: User Transparent Parallel Processing of the 2004 NIST TRECVID Data Set. In: Proceedings of IPDPS 2005, Denver, USA (April 2005)

10. Snoek, C., et al.: The Semantic Pathfinder: Using an Authoring Metaphor for Generic Multimedia Indexing. *IEEE Trans. Pat. Anal. Mach. Intel.* 28(10), 1678–1689 (2006)
11. Snoek, C., et al.: MediaMill: Exploring News Video Archives based on Learned Semantics. In: *ACM Multimedia 2005*, Singapore (November 2005)
12. Geusebroek, J., Seinstra, F.: Color-Based Object Recognition by a Grid-Connected Robot Dog. In: *AAAI 2007 - Video Competition*, Vancouver, Canada (July 2007)
13. Seinstra, F., et al.: Scalable Wall-Socket Multimedia Grid Computing, First IEEE International Scalable Computing Challenge, First Prize Winner (May 2008)
14. Golub, G., Reinsch, C.: Singular Value Decomposition and Least Squares Solutions. *Numerische Mathematik* 14(5), 403–420 (1970)
15. Evans, D., Gusev, M.: Systolic SVD and QR Decomposition by Householder Reflections. *Int. J. Comp. Math.* 79(4), 417–439 (2002)
16. Seinstra, F., et al.: Finite State Machine-Based Optimization of Data Parallel Regular Domain Problems Applied in Low-Level Image Processing. *IEEE Transactions on Parallel and Distributed Systems* 15(10), 865–877 (2004)
17. Seinstra, F., et al.: A Software Architecture for User Transparent Parallel Image Processing. *Parallel Computing* 28(7-8), 967–993 (2002)

Topic 11

Multicore and Manycore Programming

Introduction

Barbara Chapman*, Bart Kienhuis*, Eduard Ayguadé*, François Bodin*,
Oscar Plata*, and Eric Stotzer*

During the past few years, mainstream computing has rapidly embraced multi-core architectures and there is now considerable research into the design of such systems and into all aspects of their utilization. In particular, much work is needed in order to improve their programmability. Limitations on the amount of memory, bandwidth constraints, multiple layers of parallelism and heterogeneous components all introduce new challenges for application development and execution. In recognition of the importance of this architectural shift, multi-core programming was added to the collection of Euro-Par topics this year. The focus of the topic covered general-purpose multi-core programming techniques, models and languages, as well as those for multi-core embedded systems, and included related work on compilers, run-time systems, and performance and scalability studies.

The topic attracted 31 submissions covering a broad variety of research into languages, compilers, runtime libraries, algorithms and application experiences, many of which targeted heterogeneous multi-core platforms. Out of these, 12 were accepted for presentation at the conference.

In their paper “Tile Percolation: an OpenMP Tile Aware Parallelization Technique for the Cyclops-64 Multicore Processor”, researchers at the University of Delaware propose a set of OpenMP pragmas that enable the application developer to control data movement on platforms with a software-managed memory hierarchy, such as the IBM Cyclops-64. They propose tile percolation, a technique using these pragmas, and demonstrate that it improves the efficiency of OpenMP codes on the Cyclops.

Ayguade, Badia, Igual, Labarta, Mayo, and Quintana-Orti present an extension of the Star Superscalar programming model for expressing the parallelism in applications targeting a general purpose CPU connected to multiple GPGPUs in their paper “An Extension of the StarSs Programming Model for Platforms with Multiple GPUs”. Their approach requires few modifications to application code in order to adapt it to heterogeneous systems with multiple memory spaces.

“StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multi-core Architectures”, from Augonnet, Thibault, Namyst, and Wacrenier at the University of Bordeaux describes a novel runtime system that provides a unified execution model and data management library for heterogeneous systems. StarPU can support the generation of parallel tasks, as well as runtime techniques for scheduling them on heterogeneous systems in a manner that directly exploits the heterogeneity.

* Topic chairs.

XJava is a Java extension for object-oriented parallel programming. It is described, along with its implementation, in “XJava: Exploiting Parallelism with Object-Oriented Stream Programming” by Otto, Pankratius, and Tichy. It relies on tasks as a central construct for achieving parallelism and composability while preventing synchronization bugs.

Yan, Grossman, and Sarkar propose a means for Java programmers to invoke CUDA kernels in “JCUDA: A programmer-friendly interface for accelerating Java Programs with CUDA”. Their approach relies on the compiler to generate the code needed to invoke the kernels and to transfer data between host and accelerator.

“Fast and Efficient Synchronization and Communication Collective Primitives for Dual Cell-based Blades” by Gaona, Fernandez, and Acacio describes the design of collective operations that take into account the structure of heterogeneous computing systems requiring explicit data transfer between the hardware components. They show results based upon their implementation on dual Cell-based blades.

A group of researchers at Simon Fraser University and Electronic Arts Blackbox describe requirements and techniques for parallelizing video game engines in “Searching for Concurrent Design Patterns in Video Games”. They also discuss a parallel programming environment that they have created specifically targeting video game engines.

The paper entitled ”Parallelization of a Video Segmentation Algorithm on CUDA-enabled Graphics Processing Units:” presents experiences gained by Gomez-Luna, Gonzalez-Linares, Benavides and Guil using CUDA to parallelize highly compute-intensive video frame feature calculations. They compare results with an OpenMP code version running on a multi-core platform.

“A Parallel Point Matching Algorithm for Landmark Based Image Registration Using Multicore Platform” by Yang, Gong, Zhang, Nosher, and Foran reports on a new algorithm for fast point matching in the context of landmark-based medical image registration, which has near real-time requirements. They have shown significant performance improvements on a Cell platform.

The number of cores configured on future architectures is likely to be much higher than those on current platforms. In “High Performance Matrix Multiplication on Many Cores”, Yuan, Zhou, Tan, Zhang, and Fan discuss the Godson-T many-core processor and explain how they have adapted a dense matrix multiplication algorithm to exploit this architecture, achieving 97.1% of peak performance.

The paper “Parallel Lattice Basis Reduction using a Multithreaded Schnorr-Euchner LLL Algorithm” introduces a new, multithreaded variant of this algorithm which is of great importance in cryptography. The authors, Backes and Wetzel, discuss performance obtained by a Pthreads-based implementation on two dual-core Opteron processors for both sparse and dense lattice bases.

Researchers at the University of Strasbourg have developed a parallel evolutionary algorithm that exploits a GPGPU for the computationally intensive portions. The results they present in “Efficient Parallel Implementation of Evolutionary Algorithms on GPGPU Cards” are based on extensions to the EASEA specification language and compiler, and indicate up to a 100-fold performance increase over execution on a standard CPU.

Tile Percolation: An OpenMP Tile Aware Parallelization Technique for the Cyclops-64 Multicore Processor

Ge Gan¹, Xu Wang², Joseph Manzano¹, and Guang R. Gao¹

¹ Dep. of Electrical and Computer Engineering University of Delaware Newark, Delaware 19716 U.S.A.

{gan, jmanzano, ggao}@caps1.udel.edu

² Dep. of Electrical Engineering Jilin University, Jilin, China 130000
wangxu@ict.ac.cn

Abstract. Programming a multicore processor is difficult. It is even more difficult if the processor has software-managed memory hierarchy, e.g. the IBM Cyclops-64 (C64). A widely accepted parallel programming solution for multicore processor is OpenMP. Currently, all OpenMP directives are only used to decompose computation code (such as loop iterations, tasks, code sections, etc.). None of them can be used to control data movement, which is crucial for the C64 performance. In this paper, we propose a technique called *tile percolation*. This method provides the programmer with a set of OpenMP pragma directives. The programmer can use these directives to annotate their program to specify *where* and *how* to perform data movement. The compiler will then generate the required code accordingly. Our method is a semi-automatic code generation approach intended to simplify a programmer's work. The paper provides (a) an exploration of the possibility of developing pragma directives for semi-automatic data movement code generation in OpenMP; (b) an introduction of techniques used to implement tile percolation including the programming API, the code generation in compiler, and the required runtime support routines; (c) and an evaluation of tile percolation with a set of benchmarks. Our experimental results show that tile percolation can make the OpenMP programs run on the C64 chip more efficiently.

1 Introduction

OpenMP [1] is the *de facto* standard for writing parallel programs on shared memory multiprocessor system. For the IBM Cyclops-64 (C64) processor [2][3], OpenMP is one of the top selected programming model. As shown in Figure 1(a), the C64 chip has 160 homogeneous processing cores. They are connected by a 96-port, 7-stage, non-blocking on-chip crossbar switch [4]. The chip has 512KB instruction cache but no data cache. Instead, each core contains a small amount of SRAM (5.2MB in total) that can be configured into either Scratchpad Memory (SPM), or Global Memory (GM), or both in combination. Off-chip DRAM are attached onto the crossbar switch through 4 on-chip DRAM controllers. All memory modules are in the same address space and can be accessed directly by all processing cores [5]. However, different segment of the memory address space has different access latency and bandwidth. See Figure 1(b)

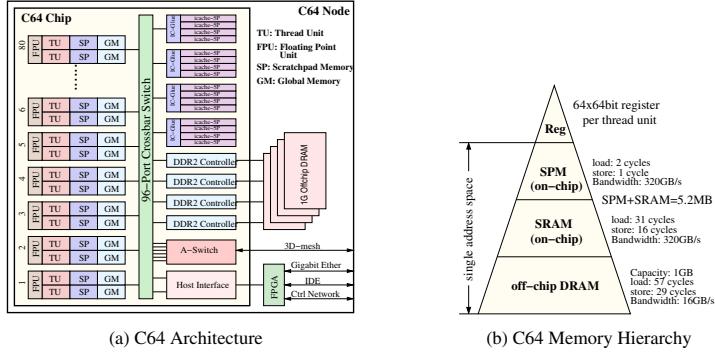


Fig. 1. The IBM Cyclops-64 Multicore Processor

for the detailed parameters of the C64 memory hierarchy. Roughly speaking, the C64 chip is a single-chip shared memory multiprocessor system. Therefore, it is easy to land OpenMP on the C64 chip [6]. However, due to its *software-managed* memory hierarchy, making an OpenMP program run efficiently on the C64 chip is not a trivial task.

Given a processor like C64, it is important for the OpenMP programs to fully utilize the on-chip memory resources. This requires the programmer to insert code in the program to move data back and forth between the on-chip and off-chip memory. Thus, the program can benefit from the short latencies of the on-chip memory and the huge on-chip bandwidth. Unfortunately, this would make the C64 multicore processor more difficult to program. From the OpenMP methodology, we have learned that it would be very helpful if we could annotate the program with a set of OpenMP pragma directives to specify where data movement is beneficial and possible, and let the compiler generate the required code accordingly. This is just like using the `parallel for` directive to annotate a loop and let the OpenMP compiler generate the multithreaded code. This would free the programmer from writing tedious data movement code.

In this paper, we introduce *tile percolation*, a tile aware parallelization technique [7] for the OpenMP programming model. The philosophy behind the tile aware parallelization technique is to enable OpenMP programmers not only the capability to direct the compiler to perform computation decomposition, but also the power to direct the compiler to perform data movement. The programmer will be provided with a set of simple OpenMP pragma directives. They can use these directives to annotate their program to instruct the compiler *where* and *how* data movement will be performed. The compiler will generate the correct computation and data movement code based on these annotations. At runtime, a set of routines will be provided to perform the dynamic data movement operations. This not only makes the programming on the C64 chip easier, but also makes sure that the data movement code inserted into the program is optimized. Since the major data objects being moved are "sub-blocks" in the multi-dimensional array, this approach is termed *tile percolation*. The major contributions of the paper are:

1. As far as the authors are aware, this is the first paper that explores the possibility of using pragma for data movement code generation in OpenMP.
2. The paper has introduced the techniques used to implement tile percolation, including the programming API, the code generation in compiler, and the required runtime support.
3. We have evaluated tile percolation with a set of benchmarks. Our experimental results show that tile percolation can make the OpenMP programs run on the C64 chip more efficiently.

The rest of the paper is organized as follows. In Section 2, we use a motivating example to show why tile percolation is necessary. Section 3 will discuss how to implement tile percolation in the OpenMP compiler. We present our experimental data and conclusions in Section 4.

2 Motivation

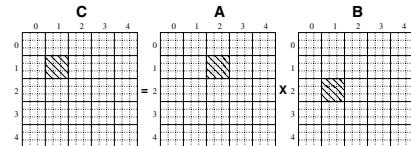
In this section, we use the tiled matrix multiplication code as a motivating example to demonstrate why writing efficient OpenMP program for the software-managed C64 memory hierarchy is not trivial and why a semi-automatic code generation approach is necessary.

```

1 for (ii=0; ii<n; ii+=b)           controlling loops
2   for (jj=0; jj<n; jj+=b)
3     for (kk=0; kk<n; kk+=b)
4       for (i=ii; i<min(ii+b,n); i++)  tiling
5         for (j=jj; j<min(jj+b,n); j++)
6           for (k=kk; k<min(kk+b,n); k++)
7             C[i][j][k] += A[i][k] * B[k][j]

```

(a) Tiled Matrix Multiplication Code



(b) Data Access Pattern in the Tiling Loops

Fig. 2. Tiled Matrix Multiplication: $C = A \times B$

Figure 2 shows the tiled matrix multiplication code and the data access pattern of the kernel loops. On the C64 chip, to make sure that this program fully utilizes the on-chip memory resources, the programmers need to insert tile movement code manually in the source code to move data tiles back and forth between the on-chip and off-chip memory. Figure 3 shows the examples of the manually inserted code. In both examples, no matter how the computations in the *controlling loops* are decomposed among the cores, for the tiling loops, small data tiles are moved into the on-chip SRAM memory and the associated computations are performed there. Figure 3(a) shows the naive version, in which the array elements are copied into the on-chip memory one by one. A better version is shown in Figure 3(b), which utilizes the off-chip memory bandwidth more efficiently. In both versions, the programmers need to study the original source code carefully to figure out how to write correct and efficient data movement code. They are forced to deal with the convoluted array index calculation, which makes their work more complicated.

A simpler approach is to let the compiler to generate the required data movement code automatically. In [8,9], the authors present their implementation of this idea on

```

0 /* allocate on-chip memory */
1 AA=(float *)sram_malloc(...);
2 BB=(float *)sram_malloc(...);
3 CC=(float *)sram_malloc(...);
4 ...
5 /* item-by-item memory copy */
6 for (i=ii; i<min(ii+b,n); i++)
7 for (j=jj; j<min(jj+b,n); j++)
8 for (k=kk; k<min(kk+b,n); k++) {
9   AA[i-ii][k-kk] = A[i][k];
10  BB[k-kk][j-jj] = B[k][j];
11  CC[i-ii][j-jj] = C[i][j];
12 }
13 /* MxM performed on-chip */
14 for (i=0; i<min(b,n-ii); i++)
15 for (j=0; j<min(b,n-jj); j++)
16 for (k=0; k<min(b,n-kk); k++)
17 CC[i][j] += AA[i][k]*BB[k][j];
18 /* copy out the results */
19 for (i=ii; i<min(ii+b,n); i++)
20 for (j=jj; j<min(jj+b,n); j++)
21 for (k=kk; k<min(kk+b,n); k++)
22 C[i][j] = CC[i-ii][j-jj];

```

(a) A naive version

```

0 /* allocate on-chip memory */
1 AA=(float *)sram_malloc(...);
2 BB=(float *)sram_malloc(...);
3 CC=(float *)sram_malloc(...);
4 ...
5 /* mcpy: optimized memory copy routine */
6 for (i=ii; i<min(ii+b,n); i++)
7 mcpy(&CC[i-ii][0], &C[i][jj], min(b,n-jj));
8 for (k=kk; k<min(kk+b,n); k++)
9 mcpy(&BB[k-kk][0], &C[k][jj], min(b,n-jj));
10 for (i=ii; i<min(ii+b,n); i++)
11 mcpy(&AA[i-ii][0], &A[i][kk], min(b,n-kk));
12 ...
13 /* on-chip calculation */
14 for (i=0; i<min(b,n-ii); i++)
15 for (j=0; j<min(b,n-jj); j++)
16 for (k=0; k<min(b,n-kk); k++)
17 CC[i][j] += AA[i][k]*BB[k][j];
18 ...
19 /* copy out the results */
20 for (i=ii; i<min(ii+b,n); i++)
21 mcpy(&C[i][jj], &CC[i-ii][0], min(b,n-jj));
22 ...

```

(b) An optimized version

Fig. 3. Examples of Manually Inserted Data Movement Code (Pseudo Code)

the IBM CELL processor. However, it is revealed that the performance of the automatically generated code is not as good as the performance of the manually reformed code. The reason is not because the compiler can not generate the optimal code, but because the static analysis performed by the compiler is not powerful enough to capture all the beneficial cases (which is a well-told story). This motivates us to develop a novel semi-automatic approach: the programmer specifies the places in their program where efficient data movement is needed, while the compiler generates the required high quality code accordingly.

3 Tile Percolation

In this section, we will use a simple example to demonstrate how to implement tile percolation. It includes three parts: the programming API, the data movement code generation in the compiler, and the required runtime support.

3.1 Programming API

In the design of the programming API for tile percolation, the following criteria should be considered. First, it must be very simple and easy to use. Second, it must be general enough to capture most of the common cases that can benefit from tile percolation. Third, it should not bring much complexity to code generation and should also not cause too much runtime overhead. According to these criteria, the tile percolation programming API is designed as OpenMP pragma directives, shown in Figure 4(a).

The tile percolation API has two new pragma directives: the `percolate` directive and the `tile` directive. The `percolate` directive specifies a *percolation region*, which

```
#pragma omp percolate [tile ...]
#pragma omp tile ro (A[jdim(A), adim(A), Ldim(A)]..[j2, a2, L2][j1, a1, L1], ...)
          wo (B[kdim(B), bdim(B), Mdim(B)]..[k2, b2, M2][k1, b1, M1], ...)
          rw (C[ldim(C), cdim(C), Ndim(C)]..[l2, c2, N2][l1, c1, N1], ...)
```

(a) The definition of the tile percolation API

percolate:	Directive name. It specifies a percolation region
tile:	Directive name. It specifies a tile region and the tile descriptors
ro:	Clause name. It specifies the tiles that are read-only in the current percolation region.
wo:	Clause name. It specifies the tiles that are write-only in the current percolation region.
rw:	Clause name. It specifies the tiles that are read and written in the current percolation region.
A, B, C:	Name of the host multi-dimensional data array
j_i, k_i, l_i :	The index variable of the for loop that defines the i_{th} dimension of the tile
a_i, b_i, c_i :	Blocking size of the i_{th} dimension of the host multi-dimensional array (i.e. A, B, and C).
L_i, M_i, N_i :	Size of the i_{th} dimension of the host multi-dimensional array
dim(..):	The dimension of the multi-dimensional array

(b) The explanation of the tile percolation API

Fig. 4. The OpenMP API for tile percolation (C/C++)

is a block of code. At the beginning of the percolation region, on-chip storage will be reserved for all data tiles that will be percolated into the on-chip memory. Then, all or some of the data tiles accessed in the percolation region will be moved into the on-chip memory and the corresponding computations will be performed there. At the end of the percolation region, data tiles that contain the results of the computations are written back to the off-chip memory (if necessary) and the reserved on-chip memory are freed.

The tile directive, on the other hand, provides the detailed information (type, shape, size, etc.) of the data tiles that will be percolated into the on-chip memory. It is always contained in a percolation region. The tile directive specifies a *tile region*, in which there is a set of `for` loops delimiting the bounds of the data tiles. In the tile directive, following the key word `tile` is a list of *tile descriptors*. The tile descriptors are divided into three groups by the key words `ro`, `wo`, and `rw`, which are the *clause* names that identify *read-only*, *write-only*, and *read-write* data tiles. At the beginning of the percolation region, data tiles specified in the `rw` clause will be copied from the off-chip memory into the on-chip memory (after the on-chip memory allocation). At the end of the percolation region, data tiles specified in the `rw` and `wo` clauses are copied back to their home locations in the off-chip memory. For data tiles specified in the `ro` clause, they will be copied into the on-chip memory at the place where the `ro` clause is specified. They will not be copied back to the off-chip memory at the end of the percolation region. The associated code in the percolation region are adjusted to access the on-chip data tiles in the computations.

The format of the tile descriptor is similar to the declaration of a multi-dimensional array variable, except that each of the tile descriptor's *dimension specifier* is a 3-tuple, not a singleton. The tile descriptor tells the compiler how the data tile is carved out from the multi-dimensional data array that hosts it. To make the paper easy to follow, we call the multi-dimensional data array that hosts the current data tile as its *host array*. The tile descriptor contains the complete information of the host array. Therefore, the number of dimension specifiers in the tile descriptor is the same as the dimension of the host array. It is not necessarily the same as the dimension of the data tile.

For a dimension specifier $[j_i, a_i, L_i]$ (see Figure 4(a)), " L_i " is the size of the i_{th} dimension of the host array (not the data tile). " a_i " is the blocking/tiling size of the i_{th} dimension of the host array. This parameter is used to carve out the data tile from its host array. Normally, if the dimension of the data tile is the same as the dimension of the host array, " j_i " is the index variable of a **for** loop in the tile region that traverse the i_{th} dimension of the data tile. If the dimension of the data tile is smaller than its host array, the element j_i in some dimension specifiers becomes trivial. Currently, we force the programmers to put a "*" there as a place holder to let the compiler know that the current dimension of the host array has been squashed away in the dimension space of the data tile. An intuitive example of this case is the expression $A[0][i][j]$ guarded by loop i and loop j . It actually represents a 2-D plane, although the expression has 3 dimension specifiers.

The tile descriptor functions like a template and the associated **for** loops instantiate this template. To make the code generation easy, currently, a writable tile descriptor (specified in the **rw** or the **wo** clause) can only has one instantiation. The read-only tiles (specified in the **ro** clause) can have multiple instantiations. Example is given in Figure 5. To put it in a simple way, roughly, the percolate directive and the tile directive tell the compiler *where* the data tiles will be percolated and the tile descriptors tell the compiler *how* the data tiles are percolated. The code example that shows the usage of the tile percolation API is in Figure 5. The detailed explanation will be presented in the next sub-section.

3.2 Code Generation

The code in Figure 5 shows how to use the tile percolation API. The pragma at line 1 is the canonical **parallel for** directive that specifies how the computation iterations are distributed among the parallel threads. The pragma at line 5 is a percolate directive and line 8 is a tile directive. The percolate directive specifies a percolation region, from line 6 to 16. The tile directive specifies a tile region, from line 10 to 15, in which there are there data tiles, represented by " $A[i, b, n][k, b, n]$ ", " $B[k, b, n][j, b, n]$ ", and " $C[i, b, n][j, b, n]$ ". The first two tiles are read-only and the last one is both readable and writable in the current percolation region. They direct the compiler to generate the correct data percolation and computation code.

The tile descriptor " $C[i, b, n][j, b, n]$ " specifies a data tile contained in the host array C , a 2D $n \times n$ matrix. In this tile descriptor, " C " provides the name of the host array, which also tells the compiler the type of the data element of the tile. " n " in the dimension specifier tells the size of the each dimension of the host array. " b " reveals how the matrix is tiled. " i " and " j " are two index variables that inform the compiler that the **for** loops at line 11 and 12 are used to construct the data tile. Since the lower and upper bounds of " i " and " j " are fixed in the current percolation region, there is only one instantiation for this tile template (i.e. descriptor). The clause name "**rw**" indicates that this data tile will be read and written in the current percolation region. So, it will be copied into the on-chip memory at line 6, where the percolation region starts. It will also be copied out to off-chip memory at line 16, where the percolation region ends.

Similarly, data tile " $A[i, b, n][k, b, n]$ " and " $B[k, b, n][j, b, n]$ " are contained in host array " A " and " B ", which are also 2D $n \times n$ matrix. Since both of them

```

1 #pragma omp parallel for collapse(2)
2 for (ii=0; ii<n; ii+=b)
3   for (jj=0; jj<n; jj+=b)
4   {
5     #pragma omp percolate
6     {
7       for (kk=0; kk<n; kk+=b)
8         #pragma omp tile ro (A[i,b,n][k,b,n],B[k,b,n][j,b,n]) \
9             rw (C[i,b,n][j,b,n])
10        {
11          for (i=ii; i<min(ii+b,n); i++)
12            for (j=jj; j<min(jj+b,n); j++)
13              for (k=kk; k<min(kk+b,n); k++)
14                C[i][j] += A[i][k]*B[k][j];
15        }
16      }
17    }

```

Fig. 5. Pseudo Code of the Tile Percolation Example

are read-only data tiles, they are copied into the on-chip memory at line 8, where they are specified in the `ro` clause. They do not need to be copied back to the off-chip memory at the end of the percolation region. Because the lower and upper bounds of "k" are changing (line 7), as we may notice, there are multiple instantiations for these two tile descriptors. All instantiations of the same data tile will reuse the same memory block allocated to it. The example is shown in Figure 6.

Figure 6 presents the code generated for the tile percolation program in Figure 5. First, it allocates on-chip memory for all three data tiles (line 5 to 7). This is achieved by calling the runtime routine `_sram_malloc`, which is inserted in by the compiler. The size of the data tile is calculated by multiplying each of its dimension size, which is obtained from its blocking size. This guarantees that the memory block allocated is big enough to hold the corresponding data tile. If the memory allocations succeed, the read-write data tiles will be copied into the on-chip memory by calling the runtime library routine `_copy2Don` (line 16). Otherwise, no data movement happens and the program execution jumps to the original code (line 12), where computations are performed on off-chip data tiles (line 36).

The other two read-only data tiles are percolated into the on-chip memory between the `for` loops at line 18 and 25. This location corresponds to the place in the original code where they are specified in the `ro` clause. The `for` loops between line 25 and 28 perform matrix multiplication on "`_AA[][]`", "`_BB[][]`", and "`_CC[][]`", which are all located in the on-chip memory. After one kernel computation (line 25 to 28) is finished, the new instantiation of "`_AA[][]`" and "`_BB[][]`" are copied from the off-chip memory into the on-chip memory and are stored in the same memory block. Then it begins the next iteration. Before exiting the percolation region, the `rw` data tile "`_CC[][]`" is copied back to its home location in the off-chip memory (line 32). Meanwhile, the on-chip memory storage allocated to all the percolated data tiles are freed.

To generate the code like in Figure 6, the compiler needs to handle three tasks: (1) generate code for managing on-chip memory; (2) generate code for managing memory copy; (3) adjust the computation code to access on-chip data tiles. We leave the discussion of the first two items to the next sub-section, because they are mostly related to the runtime. Here, we focus on the third problem.

```

1  {
2    /* Enter the percolation region.
3     * Allocate on-chip memory for all data tiles
4     */
5    _CC=(float *)_sram_malloc(sizeof(float)*b*b);
6    _AA=(float *)_sram_malloc(sizeof(float)*b*b);
7    _BB=(float *)_sram_malloc(sizeof(float)*b*b);
8
9    if (_CC==NULL || _AA==NULL || _BB==NULL)
10   {
11     _sram_free(_AA); _sram_free(_BB); _sram_free(_CC);
12     goto orig;
13   }
14
15  /* Copy "rw" data tiles from off-chip memory to on-chip memory */
16  _copy2Don(sizeof(float),_CC,&C,n,n,ii,jj,min(b,n-ii),min(b,n-jj));
17
18  for (kk=0; kk<n; kk+=b)
19  {
20    /* Copy "ro" data tiles from off-chip memory to on-chip memory */
21    _copy2Don(sizeof(float),_AA,&A,n,n,ii,kk,min(b,n-ii),min(b,n-kk));
22    _copy2Don(sizeof(float),_BB,&B,n,n,kk,jj,min(b,n-kk),min(b,n-jj));
23
24    /* on-chip calculation */
25    for (i=0; i<min(b,n-ii); i++)
26      for (j=0; j<min(b,n-jj); j++)
27        for (k=0; k<min(b,n-kk); k++)
28          _CC[i][j]+=_AA[i][k]*_BB[k][j];
29  }
30
31  /* copy out the results back to off-chip memory */
32  _copy2Doff(sizeof(float),_CC,&C,n,n,ii,jj,min(b,n-ii),min(b,n-jj));
33  _sram_free(_AA); _sram_free(_BB); _sram_free(_CC);
34  goto out;
35
36 orig:
37    /* Original code with out percolation */
38    ...
39 out:
40 }

```

Fig. 6. Code generation example for tile percolation (Pseudo Code)

Adjusting the computation code to access on-chip data tiles includes two sub-tasks: **(i)** calibrate the lower and upper bounds for each `for` loop that is involved in traversing the elements of the data tile; **(ii)** update the tile access expressions accordingly. These tasks are easy because the data tile is copied as one 2D array from its home location (in which, the elements are physically scattered in memory) into a piece of physically contiguous memory block (in which, the elements are consecutive). We only need to know the base address of the memory block and the size of each dimension of the data tile. The value of the tile's dimension size can be easily obtained from its tile descriptor. The base address of the memory block that assigned to the current data tile can be obtained from the corresponding runtime function call (`_sram_malloc`). With this information, it is easy for the compiler to generate the correct code. Most of time, we just perform a kind of simple 1-to-1 replacement. For example, the new lower bound of a `for` loop is always set to zero and the new upper bound is calculated by subtracting the old lower bound from the old upper bound.

3.3 Runtime Routines and Runtime Support

As we have mentioned in the last section, the tile percolation runtime needs to handle the on-chip memory allocation and the memory copy for the percolated data tiles. We provide a set of routines (with clear interface) in the runtime library for the compiler. The compiler, accordingly, would insert the required runtime function calls in the program during code generation.

The runtime routines `_sram_malloc` and `_sram_free` are responsible for allocating on-chip memory for the percolated data tiles. To allocate the correct memory storage for the tile, we need to know three values: (i) the number of dimensions of the tile; (ii) the size of each dimension; and (iii) the type of each data element. The number of dimensions of the tile is the number of non-trivial dimension specifiers in its tile descriptor. The dimension size is always set to the blocking size. This guarantees that the allocated memory block is big enough to hold any instantiation of the tile descriptor. The type of the data element is obtained from the name of the tile descriptor.

For each percolation region, the "all-or-none" policy is adopted in memory allocation. The program either continues execution after *all* of its memory allocation requests were satisfied, or, if *any* of its memory allocation request failed, it jumps to the original code to perform the computations on the off-chip data tiles. Because the compiler guarantees that all memory allocations occur at the beginning of the percolation region and all memory frees occur at the end of the percolation region, the memory allocation failure would not cause dead lock among the concurrent OpenMP threads. This greatly simplifies design of the runtime support and also simplifies code generation in compiler.

For the memory copy task, we provide the set of runtime routines presented in Figure 7. Currently, we support tile percolation for 1D-, 2D-, and 3D-array. They can cover most of the practical cases. Each kind of multi-dimensional array has its own

```
_copy1Don(sz,_on,_off,D1,x,b1)
_copy1Doff(sz,_on,_off,D1,x,b1)
_copy2Don(sz,_on,_off,D1,D2,x,y,b1,b2)
_copy2Doff(sz,_on,_off,D1,D2,x,y,b1,b2)
_copy3Don(sz,_on,_off,D1,D2,D3,x,y,z,b1,b2,b3)
_copy3Doff(sz,_on,_off,D1,D2,D3,x,y,z,b1,b2,b3)
...
```

(a) The runtime routines for memory copy

<code>_copy[1D 2D 3D] on:</code>	Runtime routines that copy the off-chip data tile into the on-chip memory;
<code>_copy[1D 2D 3D] off:</code>	Runtime routines that copy the on-chip data tile back to the off-chip memory;
<code>sz:</code>	Size of the element of the data tile;
<code>_on:</code>	The address of the on-chip memory block used to hold the percolated data tile;
<code>_off:</code>	The address of the home location of the percolated data tile in the off-chip memory;
<code>D1, D2, D3:</code>	The size of each dimension of the percolated data tile, from the lowest dimension to the highest dimension;
<code>x, y, z:</code>	Position of the percolated data tile in the host array. It is represented by the coordinate of its first element in the host array, from the lowest dimension to the highest dimension;
<code>b1, b2, b3:</code>	Blocking size of each dimension of the host array, from the lowest dimension to the highest dimension;

(b) The explanation of the runtime routines

Fig. 7. The runtime routines for on-chip and off-chip memory copy

memory-copy routines (see Figure 7(a)). The routines with the suffix “*on*” are used to copy data tiles from off-chip memory to on-chip memory, while the routines with the suffix “*off*” are used to copy data tiles from on-chip memory to off-chip memory. The parameters that are required in the address calculation in these memory-copy routines are supplied in the argument list. We use the “long argument list” instead of the “packed argument structure” because we try to avoid inserting unnecessary dynamic memory allocation function calls in code generation. We feel that generating dynamic memory allocation code is tricky and error-prone.

According to our design, there are some assumptions on the on-chip and the off-chip data tiles. For the on-chip data tile, it must reside in a contiguous memory block. For the off-chip data tile, it must be a sub-block of a multi-dimensional array and the multi-dimensional array must also reside in a contiguous memory block. Because the percolated data tile is only a sub-block in its host array, its memory layout is not contiguous. Physically, it consists of many data strips (or rows) that are separated by an equal distance. Hence, the parameters provided in the argument list should be able to be used to calculate the start address and the size of each data strip in the tile. With the above assumptions, it is easy to interpret the argument list of the memory-copy routines. For example, the routine `_copy2Don` copies a 2D data tile from off-chip memory to on-chip memory. The argument “`_off`” gives the start address of its host array, (i.e. the address of the first element). “`D1`” and “`D2`” tell the dimension size of the array. “`x`” and “`y`” specify the coordinates of the data tile in its host array. “`b1`” and “`b2`” reveal the blocking size of each dimension of the host array. “`b1`” and “`b2`” are the default size (of each dimension) of the percolated data tile. To handle the corner cases, the routine will calculate the effective size at runtime. The size of the data tile element is shown in “`sz`”. With the above information, it is easy for the runtime routine to calculate the address and size of each data strip and copy it around with the optimized library code. All these arguments are provided in the tile percolation directives and can be easily extracted out by the compiler.

In essence, the arguments listed above characterize the position and the size of a data tile and its host array accurately. It doesn’t matter whether this data tile and its host array are real multi-dimensional array (in the language sense) or not. As long as all the array access expression are affine functions of the loop indices, they can be declared (physically) as an 1D array but accessed by the programmers (logically) as a multi-dimensional array. The compiler will take care of the convoluted indices calculation.

4 Experiments

We evaluated tile percolation with four scientific kernels (SAXPY, SASUM, SGEMV, and SGEMM) [10] and two NAS benchmarks (EP and MG). The tile percolation was implemented through source-to-source program transformation and was prototyped in the Omni compiler [11]. The experiments were conducted on the FAST simulator [2], an execution-driven and binary-compatible C64 simulator with accurate instruction timing. Figure 11(b) gives the detailed latency numbers of the load/store operations when accessing different memory segments. The preliminary experiment results are shown in Figure 8. Due to the space limit, we only present the speedup of each testcase.

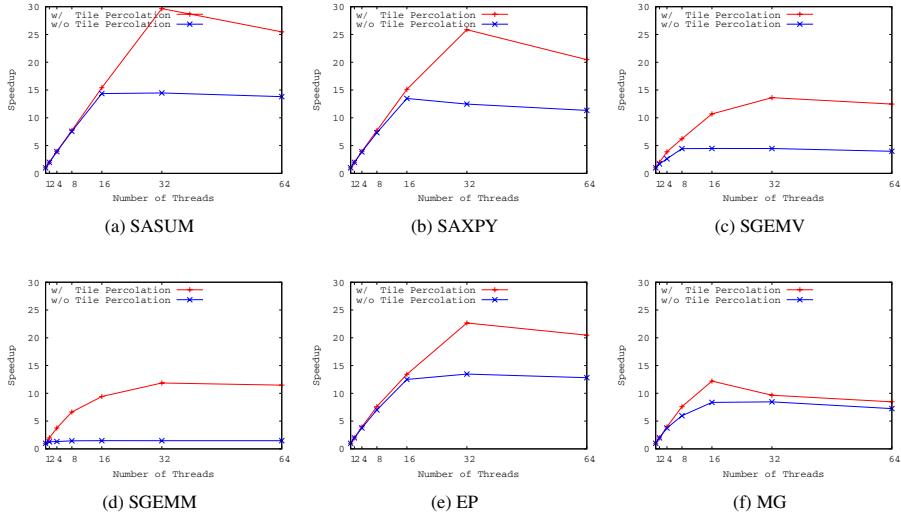


Fig. 8. Experiment Results: Comparison of Speedup

After applying tile percolation, the speedup of all testcases get significant improvement. The greatest improvement happens on SGEMM¹. This testcase has $O(n^3)$ floating-point operations but only access $O(n^2)$ data. However, without reusing the data that have been brought into on-chip memory by the previous computations, the program has very poor scalability. Because it would have $O(n^3)$ number of memory accesses going into the off-chip memory. This would quickly exhaust the off-chip bandwidth. Without using on-chip memory, its diminishing return is 2-thread. After applying the tile percolation optimization, the number of memory accesses has been reduced to $O(n^2(1 + 2n/b))$. Its speedup increased from less than 4 to around 12. For other testcases, their floating-point computations are $O(n^2)$ (SGEMV) or $O(n)$ (EP). So their speedup enhancement is not as big as SGEMM.

An interesting finding is that, without applying tile percolation, most testcases' speedup diminishing return point is at 16-thread. They are SASUM, SAXPY, EP, and MG. The speedup diminishing return point of SGEMV is 8-thread, while for SGEMM, it is 2-thread. For SASUM, its memory accesses and floating-point operations are the same. This reveals that, without on-chip data reuse, the off-chip bandwidth would be saturated when there are more than running 16 threads.

Acknowledgments

This work was supported by NSF (CNS-0509332, CSR-0720531, CCF-0833166, CCF-0702244), and other government sponsors. We thank all the members of CAPSL group at University of Delaware. We thank Jason Lin and Lei Huang for their valuable comments and feedback.

¹ We use 256×256 matrix, the data tile is 16×16 .

References

1. OpenMP Architecture Review Board: OpenMP Application Program Interface Version 3.0 (May 2008), <http://www.openmp.org/mp-documents/spec30.pdf>
2. del Cuvillo, J., Zhu, W., Hu, Z., Gao, G.R.: Fast: A functionally accurate simulation toolset for the cyclops-64 cellular architecture. In: Workshop on Modeling, Benchmarking and Simulation (MoBS 2005) of ISCA 2005, Madison, Wisconsin (June 2005)
3. del Cuvillo, J., Zhu, W., Hu, Z., Gao, G.R.: Towards a software infrastructure for cyclops-64 cellular architecture. In: HPCS 2006, Labroda, Canada (June 2005)
4. Zhang, Y., Jeong, T., Chen, F., Wu, H., Nitzsche, R., Gao, G.R.: A study of the on-chip interconnection network for the ibm cyclops64 multi-core architecture. In: IPDPS 2006: Proceedings of the 20th International Parallel and Distributed Processing Symposium, Rhodes Island, Greece, April 25-29 (2006)
5. Hu, Z., del Cuvillo, J., Zhu, W., Gao, G.R.: Optimization of dense matrix multiplication on ibm cyclops-64: Challenges and experiences. In: Euro-Par 2006, Parallel Processing, 12th International Euro-Par Conference, Dresden, Germany, Proceedings, August 28 - September 1, 2006, pp. 134–144 (2006)
6. del Cuvillo, J., Zhu, W., Gao, G.: Landing openmp on cyclops-64: an efficient mapping of openmp to a many-core system-on-a-chip. In: CF 2006: Proceedings of the 3rd conference on Computing frontiers, pp. 41–50. ACM, New York (2006)
7. Gan, G., Wang, X., Manzano, J., Gao, G.R.: Tile reduction: the first step towards openmp tile aware parallelization. In: OpenMP in a New Era of Parallelism, IWOMP 2009, International Workshop on OpenMP. Springer, Heidelberg (2009)
8. Chen, T., Zhang, T., Sura, Z., Tallada, M.G.: Prefetching irregular references for software cache on cell. In: CGO 2008: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization, pp. 155–164. ACM, New York (2008)
9. Chen, T., Lin, H., Zhang, T.: Orchestrating data transfer for the cell/b.e. processor. In: ICS 2008: Proceedings of the 22nd annual international conference on Supercomputing, pp. 289–298. ACM, New York (2008)
10. Tarditi, D., Puri, S., Oglesby, J.: Accelerator: using data parallelism to program gpus for general-purpose uses. In: ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems, pp. 325–335. ACM, New York (2006)
11. Kusano, K., Satoh, S., Sato, M.: Performance evaluation of the omni openMP compiler. In: Valero, M., Joe, K., Kitsuregawa, M., Tanaka, H. (eds.) ISHPC 2000. LNCS, vol. 1940, pp. 403–414. Springer, Heidelberg (2000)

An Extension of the StarSs Programming Model for Platforms with Multiple GPUs

Eduard Ayguadé¹, Rosa M. Badia^{1,2}, Francisco D. Igual³, Jesús Labarta¹,
Rafael Mayo³, and Enrique S. Quintana-Ortí³

¹ Barcelona Supercomputing Center – Centro Nacional de Supercomputación (BSC–CNS) and Universitat Politècnica de Catalunya, Nexus II Building, C. Jordi Girona 29, 08034–Barcelona, Spain

{eduard.ayguade,rosa.m.badia,jesus.labarta}@bsc.es

² Consejo Superior de Investigaciones Científicas (CSIC), Spain

³ Depto. de Ingeniería y Ciencia de Computadores, Universidad Jaume I (UJI), 12.071–Castellón, Spain

{figual, mayo, quintana}@icc.uji.es

Abstract. While general-purpose homogeneous multi-core architectures are becoming ubiquitous, there are clear indications that, for a number of important applications, a better performance/power ratio can be attained using specialized hardware accelerators. These accelerators require specific SDK or programming languages which are not always easy to program. Thus, the impact of the new programming paradigms on the programmer’s productivity will determine their success in the high-performance computing arena. In this paper we present GPU Superscalar (GPUs), an extension of the Star Superscalar programming model that targets the parallelization of applications on platforms consisting of a general-purpose processor connected with multiple graphics processors. GPUs deals with architecture heterogeneity and separate memory address spaces, while preserving simplicity and portability. Preliminary experimental results for a well-known operation in numerical linear algebra illustrate the correct adaptation of the runtime to a multi-GPU system, attaining notable performance results.

Keywords: Task-level parallelism, graphics processors, heterogeneous systems, programming models.

1 Introduction

In response to the combined hurdles of maximum power dissipation, large memory latencies, and little instruction-level parallelism left to be exploited, all major chip manufacturers have finally adopted multi-core designs as the only means to exploit the increasing number of transistors dictated by Amdahl’s Law. Thus, desktop systems equipped with general-purpose four-core processors and graphics processors (GPUs) with hundreds of fine-grained cores are routine today [8]. On the other hand, NVIDIA and AMD GPUs, and other accelerators like the IBM Cell B.E. or ClearSpeed boards, have demonstrated that a high performance/power ratio can be attained for certain applications using their specialized

hardware. Thus, it is natural to think that the amount of transistors available in future systems will make feasible to integrate in the chip functionalities similar to a hardware accelerator, which so far were external to the processor.

While novel and interesting heterogeneous architectures are expected for the future, we believe that it is the software (i.e., how easy is to program the new parallel architectures) that will determine their success or failure. The StarSs programming model addresses the programmability problem exploiting task-level parallelism [4][2][10]. It consists of a few OpenMP-like pragmas, a source-to-source translator, and runtime system that schedules tasks to execution preserving dependencies among tasks. Instantiations of the StarSs programming model include GRIDSS (for the Grid), CellSSs (for the Cell B.E.), and SMPSSs (for multicore processors). In this paper we extend StarSs with a new instantiation, GPUSs (GPU Superscalar), with the following specific contributions:

- *Heterogeneity*: The target architecture for GPUSs is fundamentally different: a heterogeneous system with a general-purpose processor and multiple GPUs. Although a similar approach has been investigated by some of the authors as part of the FLAME project [13], the focus there was in the specific domain of dense linear algebra. GPUSs is a general-purpose programming model, also valid for non-numerical applications.
- *Separate memory spaces*: We investigate the use of techniques such as software caches, cache coherence mechanisms, and scheduling of bundles of tasks from CellSSs and FLAME. The purpose is to hide the existence of multiple memory address spaces (as many as GPUs plus that of the general-purpose processor) from the programmer while still delivering high performance. The internal memory hierarchy of the graphics processor should be explicitly managed by the programmer of the kernels in order to attain high performance. This management is out of the scope of our runtime and programming model.
- *Simplicity*: GPUSs inherits the simplicity of StarSs, consisting of a reduced number of OpenMP-like pragmas, and the corresponding tailored versions of the StarSs translator and runtime.
- *Portability*: Although we illustrate a GPU-based implementation, most of the techniques shown can be easily applied to other type of multi-accelerator platform without major modifications of the runtime or the user codes.

Few works target the automatic parallelization of codes focusing platforms with multiple hardware accelerators. In [13] the authors propose an extension of the SuperMatrix runtime to execute linear algebra codes on systems with multiple graphics processors. Although targeting systems with one GPU, [7] presents a compiler framework for automatic source-to-source translation of OpenMP applications into optimized GPU code.

The rest of the paper is organized as follows. In Section 2 we introduce the Cholesky factorization as a motivating example. Practical aspects of an algorithm to compute this operation are also given in that section. Sections 3 and 4 contain the major contributions of this work. In the first one, we illustrate the use of the GPUSs *constructs* to parallelize a blocked algorithm for the Cholesky

factorization; in the second, we describe the main features of the GPUSs runtime which put these ideas into practice. Section 5 contains initial results for this operation on a desktop system with two Intel QuadCore processors connected to a TESLA s1070 (four NVIDIA GT200 processors). Finally, concluding remarks and future work are summarized in Section 6.

2 Computing the Cholesky Factorization

To illustrate the GPUSs extension, we will use an important operation for the solution of linear systems of equations: the Cholesky factorization. This is the first step in solving the system $Ax = b$, where $A \in \mathbb{R}^{n \times n}$ is a symmetric positive definite (SPD) matrix. While numerical linear algebra operations like the Cholesky factorization are appealing because of their use in practical applications, we note that GPUSs targets the parallelization of general numerical and non-numerical codes. Here we chose this particular operation because it exhibits tasks parallelism and an elaborated pattern of data dependencies among tasks.

The Cholesky factorization of a dense SPD matrix $A \in \mathbb{R}^{n \times n}$ is defined as

$$A = LL^T, \quad (1)$$

```

#define NT ...
#define TS ...

void Cholesky( float * A ) {
    int i, j, k;

    for ( k = 0; k < NT; k++ ) {
        chol_spotrf( A[k*NT+k] ); // Factorize diagonal block

        for ( i = k+1; i < NT; i++ )
            chol_strsm( A[k*NT+k], A[k*NT+i] ); // Triangular solves

        for ( i = k+1; i < NT; i++ ) { // Update trailing submatrix
            for ( j = k+1; j < i; j++ )
                chol_sgemm( A[k*NT+i], A[k*NT+j], A[j*NT+i] );
            chol_ssyrk( A[k*NT+i], A[i*NT+i] );
        }
    }

    int main( void ) {
        int i, j, k;

        float *A[NT][NT]; // Allocate matrix of NT x NT blocks,
                           // of dimension TS x TS each
        for ( i = 0; i < NT; i++ )
            for ( j = 0; j <= i; j++ )
                A[i][j] = ( float * ) malloc( TS*TS*sizeof( float ) );

        Init_matrix( A ); // Initialize elements of the matrix
        Cholesky( A ); // Compute the Cholesky factor
    }
}

```

Fig. 1. Blocked algorithm for computing the Cholesky factorization

```
// LAPACK spotrf wrapper
void chol_spotrf( float *A )
{
    int info = 0;
    int ts    = TS;
    spotrf( "Lower", &ts, A, &ts,
             &info );
}
```

```
// BLAS trsm wrapper
void chol_strsm( float *T,
                  float *B )
{
    float sone = 1.0;
    int ts     = TS;
    strsm( "Right", "Lower",
            "Transpose", "Non-Unit",
            &ts, &ts,
            &sone, T, &ts,
            B, &ts );
}
```

```
// BLAS gemm wrapper
void chol_sgemm( float *A,
                  float *B,
                  float *C )
{
    float sone      = 1.0,
          minus_sone = -1.0;
    int ts         = TS;
    sgemm( "NoTranspose", "Transpose",
            &ts, &ts, &ts,
            &minus_sone, A, &ts,
            B, &ts,
            &sone,      C, &ts );
}
```

```
// BLAS syrk wrapper
void chol_ssyrk( float *A,
                  float *C )
{
    float sone      = 1.0,
          sminus_one = -1.0;
    int ts         = TS;
    ssyrk( "Lower", "NoTranspose",
            &ts, &ts,
            &sminus_one, A, &ts,
            &sone,      C, &ts );
}
```

Fig. 2. Building blocks for the blocked algorithm for computing the Cholesky factorization

where $L \in \mathbb{R}^{n \times n}$ is a lower triangular matrix known as the Cholesky factor of A . (A can be decomposed as $A = U^T U$, with $U \in \mathbb{R}^{n \times n}$ being upper triangular.)

A blocked algorithm for the Cholesky factorization is given in routine `Cholesky` in Figure 1 together with the main function that allocates the matrix and invokes this routine. Following the common practice, the factorization algorithm overwrites the lower triangular part of the array A with the entries of L , while the strictly upper triangular part of the matrix remains unmodified. (Note here that both in the driver and the factorization routine, the matrix is considered to be stored by blocks; i.e., entries in the same block are stored in contiguous positions in memory. This is known to yield better performance for numerical applications; see, e.g., [95].) Figure 2 shows that `chol_spotrf`, `chol_strsm`, `chol_sgemm`, and `chol_ssyrk` are simple wrappers to routines `spotrf`, `strsm`, `sgemm`, and `ssyrk` from LAPACK (the first one) and BLAS (the last three ones) [106]. Highly tuned implementations of the BLAS routines are provided by most hardware vendors; examples of interest to this work include Intel MKL and NVIDIA CUBLAS.

3 The GPUSs Framework

3.1 Target Platform

Multi-accelerator systems are the natural target platform for the GPUSs framework. A generic multi-accelerator system consists of a general-purpose processor

(possibly with several cores) called *host*, connected to a number of hardware accelerators, or *devices*. We assume that each device can only access its own local memory space. In our case, the devices are programmable GPUs which communicate with the CPU via a PCIEexpress bus. Direct communication between devices is not possible, so that data transfer between them must be performed through the host memory (main memory).

Most modern hardware accelerators are designed as *many-core* systems, replicating specialized fine-grained cores, and featuring an internal memory hierarchy. It is important to point out that exploiting the hardware parallelism due to the existence of multiple fine-grained cores inside the accelerator is out of the scope for GPUSs. Our approach considers each accelerator as a single execution unit (or coarse-grained core), capable of efficiently executing pieces of code (or *ernels*) written by the programmer. Thus, GPUSs is not aware of the internal architecture of the hardware accelerator. It only exploits the parallelism derived from the existence of multiple hardware accelerators connected to the system.

GPUSs can be used to parallelize a code consisting of several invocations to CUDA kernels initially designed to be executed on a single GPU, adapting it to a multi-GPU system. Tuning techniques applied inside those kernels (use of shared memory, coalesced accesses to target memory, absence of bank conflicts in the access to shared memory,...) are CUDA-specific improvements which will affect the global performance of the parallel execution. However, those details are transparent to GPUSs, whose goal is to efficiently dispatch the execution of kernels to different accelerators, reducing both the number of data transfers and device idle times.

3.2 Programming Model

The programming model introduced by StarSs and extended by GPUSs allows the automatic parallelization of sequential applications. A runtime system is in charge of using the different hardware resources of the platform (the multi-core general-purpose processor and the GPUs) in parallel to execute the annotated sequential code.

It is responsibility of the programmer to annotate the sequential code to indicate that a given piece of code will be executed on a GPU. These annotations themselves do not indicate a parallel region, but a function which can be run on an accelerator. In a system with multiple accelerators, the runtime extracts the parallelism by building a data dependency graph (in which each node represents an instance of an annotated function, also called a *task*, and edges denote data dependencies between tasks) and by executing independent tasks on the different accelerators in parallel.

GPUSs basically provides two OpenMP-like constructs to annotate code. The first one, directly inherited from StarSs, is used to identify a unit of work, or task, and can be applied to tasks that are just composed of a function call, as well as to headers or definitions of functions that are always executed as tasks:

```
#pragma css task [clause-list]
{function-header | function-definition | function-call}
```

When the program calls a function annotated in this way, the runtime will create an explicit task. This construct is complemented with clauses `input`, `output`, and `inout`, which identify the directionality (input, output, or both, respectively) of the arguments to the function. The clauses are used by the GPUSs runtime to track dependencies among tasks and decide at run time whether a task can be scheduled/issued for execution.

In our particular example, we can use this construct to annotate the LAPACK and BLAS wrappers as follows:

```
#pragma css task inout ( A[TS][TS] )
void chol_spotrf( float *A );

#pragma css task input ( T[TS][TS] ) inout ( B[TS][TS] )
void chol_strsm( float *T, float *B );

#pragma css task input ( A[TS][TS] ) inout ( C[TS][TS] )
void chol_ssyrk( float *A, float *C );

#pragma css task input ( A[TS][TS], B[TS][TS] ) inout ( C[TS][TS] )
void chol_sgemm( float *A, float *B, float *C );
```

Although we have annotated the header declaration of the wrappers here, the effect is the same that would be obtained by annotating the function call or its definition. We also indicate the dimension of the non-scalar arguments in clauses; this is trivial in our example, as all matrix blocks are of size $TS \times TS$. Here TS could also be an argument to the function, with the appropriate directionality indication.

The second construct follows a recent proposal to extend the OpenMP tasking model for heterogeneous architectures in [2], and has been incorporated in GPUSs. The construct adopts the form

```
#pragma css target device( device-name ) [clause-list]
```

The `target` construct specifies that the execution of the task should be offloaded to the device specified by `device-name` (and as such, its code must be handled by the appropriate compiler back-end and its execution appropriately managed by the runtime). Tasks which are not annotated with this construct are executed on the host. For NVIDIA GPUs, the device name is `cuda`. Some additional clauses can be used with this pragma, to specify data movement between memory spaces of the shared variables inside the task. In particular

```
copy_in( data-reference-list )
copy_out( data-reference-list )
```

will move the variables in `data-reference-list` from host to device memory or vice-versa, respectively. In other words, when the corresponding task is ready to be issued, the runtime will transfer the variables in the `copy_in` list from host memory to device memory. Once the execution of the tasks is completed, variables in the `copy_out` list will be retrieved from device to host memory.

Applying the previous construct to the Cholesky factorization is simple:

```

#pragma css task inout ( A[TS][TS] )
void chol_spotrf( float *A );

#pragma css target device( cuda ) copy_in( T[TS][TS], B[TS][TS] ) \
copy_out( B[TS][TS] )
#pragma css task input ( T[TS][TS] ) inout ( B[TS][TS] )
void chol_strsm( float *T, float *B );

#pragma css target device( cuda ) copy_in( A[TS][TS], C[TS][TS] ) \
copy_out( C[TS][TS] )
#pragma css task input ( A[TS][TS] ) inout ( C[TS][TS] )
void chol_ssyrk( float *A, float *C );

#pragma css target device( cuda ) copy_in( A[TS][TS], B[TS][TS], \
C[TS][TS] ) \
copy_out( C[TS][TS] )
#pragma css task input ( A[TS][TS], B[TS][TS] ) inout ( C[TS][TS] )
void chol_sgemm( float *A, float *B, float *C );

```

In this case, we have decided to offload the execution of the BLAS wrappers `chol_strsm`, `chol_ssyrk`, and `chol_sgemm` to the GPUs. The LAPACK wrapper `chol_spotrf` will be executed on the host. The examples above use optimized implementations of these BLAS kernels for the GPU (e.g., in the NVIDIA implementation CUBLAS). Thus, we accommodate the possibility of executing a given task in the host, which may be more efficient for certain tasks.

Function definitions are not restricted to wrappers to library routine calls, as in the examples above. In GPUSs, native CUDA code could also be included in the body of the annotated functions. In this case, the programming model allows the developer to explicitly define some CUDA-specific execution parameters (basically, grid and block sizes; see [8].)

The bottom-line is that the sequential code in Figure 1 does not need to be modified to execute it in parallel on a multi-GPU platform. With the previous annotations, GPUSs automatically parallelizes the execution of the factorization, issuing tasks (or bundles of them) to the available execution units.

4 The GPUSs Runtime

4.1 Adapting the CellSs Runtime to the TESLA System

Although being very different from the architectural viewpoint, there are many conceptual similarities between the Cell and the NVIDIA TESLA architectures, which allow the adaptation of the runtime developed for the Cell to a system with multiple hardware accelerators. In general, many of these similarities also hold for other multi-accelerator systems.

Both architectures are heterogeneous: the Cell consists of a general-purpose processor, or PPE, that orchestrates execution on a set of specialized fast cores, or SPEs. All processors can communicate through a fast interconnection bus (EIB), using small memory pools for each SPE, and a global memory space accessible to all processors. On the other hand, the multi-GPU system consists

of a central general-purpose processor with a memory pool (RAM) and a number of GPUs (NVIDIA TESLA) connected to the host through a PCIExpress bus.

Some of the practical differences between both architectures have a direct implication in the design and adaptation of the runtime. First, the Cell PPE is much slower than the SPEs. Thus, the workload assigned to the PPE must be carefully designed to avoid penalizing the global performance of the runtime. In particular, critical decisions such as scheduling or resource assignation must be simplified, adopting trade-offs between simplicity and performance.

In addition, the local memory spaces for each GPU are considerably larger than the corresponding local stores of each SPE, and GPUs usually attain higher performance for large data streams [3]. This dictates the use of a coarser task granularity in GPUSs, which gives more time to the CPU to apply more elaborated scheduling techniques that may boost performance.

The third main difference lies in the interconnect between host and devices. Here, the peak bandwidth of the Cell EIB (25 GB/s per channel) is much higher than that of the PCIExpress bus (0.5 GB/s per lane in the Gen2 specification.) Moreover, direct communication between SPEs can be performed in the Cell, but this technique cannot be applied for the TESLA system where all transfers between GPUs must be done through the main memory. The penalty introduced by the use of the PCIExpress bus urges a reduction of data transfers. The impact of data transfers in multi-GPU systems has already been analyzed [13], and similar techniques can also be applied to the GPUSs runtime implementation.

4.2 The GPUSs Runtime

Despite the architectural differences between the Cell B.E. and a SMP processor, the two runtime systems developed for those architectures in the StarSs framework (CellSs and SMPSs) share many features that have been inherited by the GPUSs implementation described.

Both runtime systems are divided in two main modules. The first one is devoted to the execution of the annotated user code, task identification and generation, data renaming and scheduling. The second module manages the data movements between memory spaces (only in the CellSs implementation) and task execution. The actual implementation of those modules varies for each system; more details can be found in [12] and [4]. We will focus on the main differences and particularities introduced in the GPUSs runtime.

Similarly to the CellSs runtime (see [4][1]), three *actors* play a fundamental role in the execution of an application using the GPUSs runtime:

- A *master* thread executes the user code, intercepting calls to annotated functions, generating *tasks*, and inserting them in a Task Dependency Graph.
- A *helper* thread consumes tasks as the GPUs become idle, mapping them to the most suitable device taking into consideration data locality policies.
- A set of *worker* threads (one per GPU) which wait for available tasks, perform the necessary data transfers between RAM memory and the corresponding GPU, invoke the task call on GPU, and retrieve the results (if necessary). There is a key difference in the worker infrastructure between the Cell and

the NVIDIA TESLA system: GPUs are passive processing elements, waiting for tasks ready for execution, but without any other processing capability; thus, worker threads are executed on the CPU, while in the Cell B.E. worker threads run in the accelerator/SPEs. Therefore, the GPUSs runtime is executed entirely on the host, not divided into host and device as in the CellSs system.

As in the CellSs runtime, the main program communicates with the corresponding *worker* thread by using event signaling when a GPU becomes idle, and a new task is ready for execution; the *worker* thread receives the necessary information to identify and invoke each ready task, and locate the necessary parameters for the execution of the task. Once the task has been executed by the proper GPU and data has been transferred back to main memory, the bound *worker* thread notifies the end of the execution to the *helper* thread, which can then continue with the notification of new ready tasks. The number of *worker* threads (and thus, the number of accelerators used in the executions) can be specified by the user at runtime.

4.3 Locality Exploitation and Task Scheduling

The two classes of available memory spaces (host and device memories) can be seen as a two-level memory hierarchy: before executing a task, the bound *worker* thread transfers the necessary blocks to the local memory of its GPU, performs the computations, and transfers back the updated data.

The impact of data transfers on the basic implementation explained above can be reduced by considering the local memory space of each accelerator as a cache memory that stores recently-used data blocks. The implementation of a software cache of read-only blocks stored in the memory of each GPU can reduce the amount of data transfers between host and device memory spaces. The replacement policy (LRU in our experiments) and the number and size of cache blocks can be tuned to improve performance.

In combination with the software cache, we have implemented two different memory coherence policies to reduce the amount of data transfers:

Write-invalidate: When the execution of a task is completed, the corresponding *worker* thread invalidates the read-only copies of the blocks modified by the active GPU on the memories of the remaining GPUs, notifying the *worker* threads bound to those GPUs to do so.

Write-back: To reduce the number of transfers further, a write-back policy is employed, allowing inconsistencies between data blocks stored in the caches of the accelerators and the blocks in RAM. Data blocks written by a GPU are only updated in RAM when another GPU has to operate with them.

The system automatically handles the existence of multiple memory spaces (as many as accelerators plus the system memory space) by keeping a memory map of the cache of each accelerator. The translation of addresses between memory spaces is transparent to the user, who is not aware of the existence of several separate memory spaces in his/her code. This centralized directory, managed

by the *helper* thread, allows efficient handling of data blocks necessary for the management of the different data caches explained above. This software cache is an extension of that developed for CellSSs, in which each *worker* thread had direct control over the bound SPE local store, being implemented in a distributed way.

Additionally, the information stored in the memory directory can be used to reduce data transfers by selectively mapping tasks to the most appropriate accelerator.

5 Experimental Results

In our experiments we used an NVIDIA TESLA s1070 computing system with four NVIDIA GT200 GPUs and 16 GBytes of DDR3 memory (4 GBytes per GPU). The TESLA system is connected to a workstation with two Intel Xeon QuadCore E5440 (2.83 GHz) processors with 8 GBytes of shared DDR2 RAM memory. The Intel 5400 chipset features two PCIExpress Gen2 interfaces connected with the TESLA, which deliver a peak bandwidth of 48 Gbits/second on each interface. NVIDIA CUBLAS (version 2.2) built on top of the CUDA API (version 2.2) together with NVIDIA driver (185.18) were used in our tests. MKL 10.0.1 was employed for all computations performed in the Intel Xeon using the 8 cores available in the system. Single precision was employed in all experiments. When reporting the rate of computation, we consider the cost of the Cholesky factorization to be the standard $n^3/3$ flops (floating-point arithmetic operations), for square matrices of order n . The GFLOPS rate is computed as the number of flops divided by $t \times 10^{-9}$, where t equals the elapsed time in seconds. The cost of all data transfers between RAM and GPU memories is included in the timings. No page-locked memory has been used in the allocation of the input matrices in the host memory.

Figure 3 (left side) shows the performance results for the Cholesky factorization algorithm shown in Figure 1, executed on the TESLA s1070 system using the GPUs runtime. The four GPUs available in the TESLA system were used in the experiment, executing all the tasks exclusively on the graphics processor.

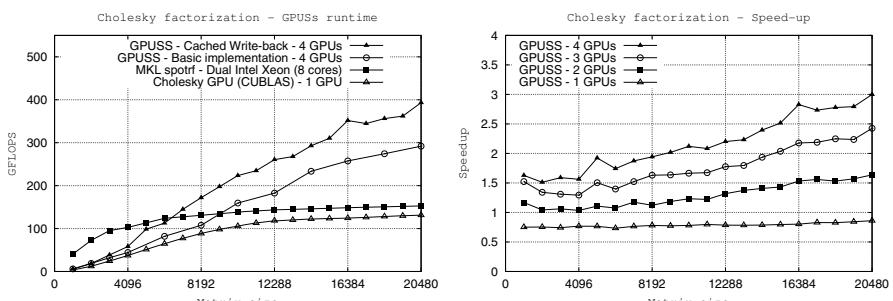


Fig. 3. Impact of the data cache and different memory coherence policies on the performance of the GPUs runtime for the Cholesky factorization (left). Speed-ups of the runtime for the Cholesky factorization using 1, 2, 3, and 4 GPUs (right).

The optimal block size, experimentally determined, is much larger in our case than in the Cell B.E. implementation in [4]: observed optimal block sizes for the TESLA system were always equal or larger than 512, while in the CellSs case the optimal block size was 64 (limited by the small size of the Local Store of each SPE, and the existence of tuned versions of the kernels only for this block sizes).

An important improvement in performance is observed in Figure 3 when the write-back policy is incorporated. The figure also displays the performances of a LAPACK-like code linked with CUBLAS executed on a single GPU [3], and the MKL multi-threaded Cholesky factorization on the eight cores of the system.

The right-hand side plot in Figure 3 reports the speed-up of the algorithm in Figure 1, with the corresponding GPU annotations, and executed using the GPUSS runtime system on 1, 2, 3, and 4 processors of the TESLA system. Speed-ups are calculated with respect to the same algorithm, linked with CUBLAS, and run on a single processor of the TESLA. The results in the figure corresponding to a single processor reveal the small overhead introduced by the runtime.

6 Concluding Remarks

In this paper we have validated the versatility of the StarSs programming model, extending it to target architectures equipped with multiple hardware accelerators. With a very small number of modifications to user's code, our approach deals with data transfers, different memory spaces, and task scheduling in a heterogeneous system. The parallelization of the codes is performed by a runtime system based on the CellSs runtime, with notable preliminary performance results for a complex operation like the Cholesky factorization.

Although the experiments and runtime have been developed for a specific multi-GPU system (NVIDIA TESLA), many of the ideas introduced here can also been applied to other architectures consisting of a workstation connected to multiple hardware accelerators via a fast interconnect.

Future work will include the implementation in the runtime of some of the extensions proposed in [2], mainly the possibility of deciding the target device (in our case, host or GPU) for the execution of a given task based on the state of the system. More complex scheduling strategies or software cache implementations, the implementation of multi-buffering to overlap transfers and computation, and ports to other multi-accelerator architectures are also in the roadmap.

Acknowledgments

The researchers at BSC-UPC were supported by the Spanish Ministry of Science and Innovation (contract no. TIN2007-60625 and CSD2007-00050), the European Commission in the context of the SARC project (contract no. 27648), the HiPEAC Network of Excellence (contract no. IST-004408), and the MareIncognito project under the BSC-IBM collaboration agreement. The researchers at UJI were supported by the Spanish Ministry of Science and Innovation/FEDER (contracts no. TIN2005-09037-C02-02 and TIN2008-06570-C04-01) and by the

Fundación Caixa-Castelló/Bancaixa (contracts no. P1B-2007-19 and P1B-2007-32). Part of this work was performed while Francisco D. Igual was visiting BSC-UPC. Support for this visit came from the *Spanish ICTS program* of the BSC.

References

1. Anderson, E., Bai, Z., Demmel, J., Dongarra, J.E., DuCroz, J., Greenbaum, A., Hammarling, S., McKenney, A.E., Ostrouchov, S., Sorensen, D.: LAPACK Users' Guide. SIAM, Philadelphia (1992)
2. Ayguadé, E., Badia, R.M., Cabrera, D., Duran, A., Gonzalez, M., Igual, F.D., Jimenez, D., Labarta, J., Martorell, X., Mayo, R., Perez, J.M., Quintana-Ortí, E.S.: A proposal to extend the OpenMP tasking model for heterogeneous architectures. In: Evolving OpenMP in an Age of Extreme Parallelism. 5th International Workshop on OpenMP, IWOMP 2009, Dresden, Germany. LNCS. Springer, Heidelberg (2009)
3. Barrachina, S., Castillo, M., Igual, F.D., Mayo, R., Quintana-Ortí, E.S.: Solving dense linear systems on graphics processors. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 739–748. Springer, Heidelberg (2008)
4. Bellens, P., Pérez, J.M., Badia, R.M., Labarta, J.: CellSs: a programming model for the Cell BE architecture. In: SC 2006: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, p. 86. ACM Press, New York (2006)
5. Chatterjee, S., Lebeck, A.R., Patnala, P.K., Thottethodi, M.: Recursive array layouts and fast matrix multiplication. IEEE Trans. on Parallel and Distributed Systems 13(11), 1105–1123 (2002)
6. Dongarra, J., Croz, J.D., Hammarling, S., Duff, I.: A set of level 3 basic linear algebra subprograms. ACM Trans. Math. Soft. 16(1), 1–17 (1990)
7. Lee, S., Min, S.-J., Eigenmann, R.: Openmp to gpgpu: a compiler framework for automatic translation and optimization. In: PPoPP 2009: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 101–110. ACM Press, New York (2009)
8. NVIDIA. NVIDIA CUDA Programming Guide 2.2 (2008)
9. Park, N., Hong, B., Prasanna, V.K.: Tiling, block data layout, and memory hierarchy performance. IEEE Trans. on Parallel and Distributed Systems 14(7), 640–654 (2003)
10. Perez, J.M., Bellens, P., Badia, R.M., Labarta, J.: CellSs: Making it easier to program the cell broadband engine processor. IBM Journal of Research and Development 51(5) (August 2007)
11. Perez, J.M., Badia, R.M., Labarta, J.: Scalar-aware grid superscalar. DAC TR UPC-DAC-RR-CAP-2006-12. Technical report, Universitat Politècnica de Catalunya, Computer Architecture Department (2006)
12. Pérez, J.M., Badia, R.M., Labarta, J.: A flexible and portable programming model for SMP and multi-cores. Technical Report 03/2007, Barcelona Supercomputing Center - CNS, Barcelona, Spain (2007)
13. Quintana-Ortí, G., Igual, F.D., Quintana-Ortí, E.S., van de Geijn, R.A.: Solving dense linear systems on platforms with multiple hardware accelerators. In: PPoPP 2009: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 121–130. ACM, New York (2009)

STARPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures

Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier

University of Bordeaux – LaBRI – INRIA Bordeaux Sud-Ouest

Abstract. In the field of HPC, the current hardware trend is to design multiprocessor architectures that feature heterogeneous technologies such as specialized coprocessors (*e.g.*, Cell/BE SPUs) or data-parallel accelerators (*e.g.*, GPGPUs).

Approaching the theoretical performance of these architectures is a complex issue. Indeed, substantial efforts have already been devoted to efficiently offload parts of the computations. However, designing an execution model that unifies all computing units and associated embedded memory remains a main challenge.

We have thus designed STARPU, an original runtime system providing a high-level, unified execution model tightly coupled with an expressive data management library. The main goal of STARPU is to provide numerical kernel designers with a convenient way to generate parallel tasks over heterogeneous hardware on the one hand, and easily develop and tune powerful scheduling algorithms on the other hand.

We have developed several strategies that can be selected seamlessly at run time, and we have demonstrated their efficiency by analyzing the impact of those scheduling policies on several classical linear algebra algorithms that take advantage of multiple cores and GPUs at the same time. In addition to substantial improvements regarding execution times, we obtained consistent *superlinear* parallelism by actually *exploiting* the heterogeneous nature of the machine.

1 Introduction

Multicore processors are now mainstream. To face the ever-increasing demand for more computational power, HPC architectures are not only going to be *massively* multicore, they are going to feature *heterogeneous* technologies such as specialized coprocessors (*e.g.*, Cell/BE SPUs) or data-parallel accelerators (*e.g.*, GPGPUs). As illustrated by the currently TOP500-leading IBM RoadRunner machine, which is composed of a mix of CELLS and OPTERONS, accelerators have indeed gained a significant audience. In fact, heterogeneity not only affects the design of computing units itself (CPU *vs* CELL’s SPU), but also memory design (cache and memory banks hierarchy) and even programming paradigms (MIMD *vs* SIMD).

But is HPC, and its usual momentum, ready to face such a revolution? At the moment, this is clearly not the case, and progress will only come from our ability to harness such architectures while requiring minimal changes to programmers’ habits. As none of the main programming standards (*i.e.*, MPI and OPENMP) currently address all the requirements of heterogeneous machines, important standardization efforts are needed. Unless such efforts are made, accelerators *will* remain a niche. In this respect,

the OpenCL initiative is clearly a valuable attempt in providing a common programming interface for CPUs, GPGPUs, and possibly other accelerators. However, the OpenCL API is a very low-level one which basically offers primitives for explicitly offloading tasks or moving data between coprocessors. It provides no support for task scheduling or global data consistency, and thus can not be considered as a true “runtime system”, but rather as a virtual device driver.

To bridge the gap between such APIs and HPC applications, one crucial step is to provide optimized versions of computation kernels (BLAS routines, FFT, and other numerical libraries) capable of running seemlessly over heterogeneous architectures. However, since no performance model that would allow to use a static hardware resource assignment is likely to emerge in the near future, these kernels must be designed to dynamically adapt themselves to the available resources and the application load.

As an attempt to provide a runtime system allowing the implementation of such numerical kernels, we have recently developed a data-management library that seamlessly enforces a coherent view of all hardware memory banks (*e.g.*, main memory, SPU local store, GPU on-board memory, etc.) [1]. This library was successfully used to implement some simple numerical kernels quite easily (using a straightforward scheduling scheme), the next step is now to abstract the concept of task on heterogeneous hardware and to provide expert programmers with scheduling facilities. Integrated within high-level programming environments such as OPENMP, this would indeed help application developers to concentrate on high-level algorithmic issues, regardless of the underlying scheduling issues.

We here propose STARPU, a simple tasking API that provides numerical kernel designers with a convenient way to execute parallel tasks over heterogeneous hardware on the one hand, and easily develop and tune powerful scheduling algorithms on the other hand. STARPU is based on the integration of the data-management facility with a task execution engine. We demonstrate the relevance of our approach by showing how heterogeneous parallel versions of some numerical kernels were developed together with advanced scheduling policies.

Section 2.2 presents the unified model we propose to design in STARPU. In section 3, we enrich our model with support for scheduling policies. We study how scheduling improves the performance of our model in section 4. We compare our results with similar works in section 5, and section 6 draws a conclusion and plans for future work.

2 The STARPU Runtime System

Each accelerator technology usually has its specific execution model (*e.g.*, CUDA for NVIDIA GPUs), and its proper interface to manipulate data (*e.g.*, DMA on the CELL). Porting an application to a new platform therefore often boils down to rewriting a large part of the application, which severely impairs productivity. Writing portable code that runs on multiple targets is currently a major issue, especially if the application needs to exploit multiple accelerator technologies, possibly at the same time.

To help tackling this issue, we designed STARPU, a runtime layer that provides an interface unifying execution on accelerator technologies as well as multicore processors. Middle layers tools (such as programming environments and HPC libraries) can

build up on top of STARPU (instead of directly using low-level offloading libraries) to keep focused on their specific role instead of having to handle efficient simultaneous use of offloading libraries. That allows programmers to make existing applications efficiently exploit different accelerators with limited effort. We now present the main components of STARPU: a high level library that takes care of transparently performing data movements, already described in a previous article [1], and a unified execution model.

2.1 Data Management

As accelerators and processors usually cannot transparently access the memory of each other, performing computations on such architectures implies explicitly moving data between the various computational units. Considering that multiple technologies may interact, and that specific knowledge is required to handle the variety of low-level techniques, in previous work [1] we designed a high level library that efficiently automates data transfers throughout heterogeneous machines. It uses a software MSI caching protocol to minimize the number of transfers, as well as partitioning functions and eviction heuristics to overcome the limited amount of memory available on accelerators.

2.2 An Accelerator-Friendly Unified Execution Model

The variety of technologies makes accelerator programming highly dependant on the underlying architecture, so we propose a uniform approach for task and data parallelism on heterogeneous platforms. We define *codelets* as an abstraction of a task (*e.g.*, a matrix multiplication) that can be executed on a core or offloaded onto an accelerator using an asynchronous continuation passing paradigm. Programmers supply implementations of *codelets* for each of the architectures that can execute them, using their respective usual programming languages (*e.g.*, CUDA) or libraries (*e.g.*, BLAS routines). An application can then be described as a set of *codelets* with data dependencies.

Declaring tasks and data dependencies. A *codelet* includes a high level description of the data and the type of access (*i.e.*, read, write or both) that is needed. Since *codelets* are launched asynchronously, this allows STARPU to reorder tasks in case this improves performance. By declaring dependencies between tasks, programmers can just let STARPU automatically enforce the actual dependencies between *codelets*.

Designing *codelet* drivers. As the aim of *codelets* is to offer a uniform execution model, it is important that it be simple enough to fit to the various architectures that might be targeted. In essence, adding the support of *codelets* for a new architecture means writing a driver that continuously does the following: request a *codelet* from STARPU, fetch its data, execute the implementation of the *codelet* for that architecture, perform its callback function and tell STARPU to unlock tasks waiting for this *codelet*'s completion. That model has been successfully implemented on top of CUDA, on multicore multiprocessors, and we are porting it to CELL's coprocessors (as we have already successfully used a similar approach in previous work in the CELL RUNTIME LIBRARY [12]).

3 A Generic Scheduling Framework for Heterogeneous Architectures

The previous section has shown how tasks can be executed on the various processing units of a heterogeneous machine. However, we did not specify how they should be distributed efficiently, especially with regards to load balancing. It should be noted that nowadays architectures have gotten so complex that it is very unlikely that writing portable code which efficiently maps tasks statically is either possible or even productive.

3.1 Scheduling Tasks in a Heterogeneous World

Data transfers have an important impact on performance, so that a scheduler favouring locality may increase the benefits of caching techniques by improving data reuse. Considering that multiple problems may be solved concurrently, and that machines are not necessarily fully dedicated (*e.g.*, when coupling codes), dynamic scheduling becomes a necessity. In the context of heterogeneous platforms, performance vary a lot according to architectures (*i.e.*, in terms of raw performance) and according to the workload (*e.g.*, SIMD code *vs.* irregular memory access). It is therefore crucial to take the specificity of each computing unit into account when assigning work.

Similarly to the problems of data transfers or task offloading, heterogeneity makes the design and the implementation of portable scheduling policies a challenging issue. So we propose to extend our uniform execution model with a uniform interface to design *codelet* schedulers. STARPU offers low level scheduling mechanisms (*e.g.*, work stealing) so that scheduler programmers can use them in a high level fashion, regardless of the underlying (possibly heterogeneous) target architecture. Since all scheduling strategies have to implement the same interface, they can be programmed independently from applications, and the user can select the most appropriate strategy at runtime.

In our model, each **worker** (*i.e.*, each computation resource) is given an *abstract* queue of *codelets*. Two operations can be performed on that queue: task submission (*push*), and request for a task to execute (*pop*). The actual queue may be shared by several workers provided its implementation takes care of protecting it from concurrent accesses, thus making it totally transparent for the *codelet* drivers. All scheduling decisions are typically made within the context of calls to those functions, but there is nothing that prevents a strategy from being called in other circumstances or even periodically.

In essence, defining a scheduling policy consists in creating a set of queues and associating them with the different workers. Various designs can be used to implement the queues (*e.g.*, FIFOs or stacks), and queues can be organized according to different topologies (*e.g.*, a central queue, or per-worker queues). Differences between strategies typically result from the way one of the queue is chosen when assigning a new *codelet* after its submission by the means of a *push* operation.

3.2 Writing Portable Scheduling Algorithms

Since they naturally fit our queue-based design, all the strategies that we have written with our interface (see Table 1) implement a greedy *list scheduling* paradigm: when a

ready task (*i.e.*, all its dependencies are fulfilled) is submitted, it is directly inserted in one of the queues, and former scheduling decisions are not reconsidered. Contrary to DAG scheduling policies, we do not schedule tasks that are not yet ready: when the last dependency of a task is executed, STARPU schedules it by the means of a call to the usual `push` function.

Restricting ourselves to list scheduling may somehow reduce the generality of our scheduling engine, but this paradigm is simple enough to make it possible to implement portable scheduling policies. This is not only transparent for the application, but also for the drivers which request work. This simplicity allows people working in the field of scheduling theory to branch higher level tools to use STARPU as an experimental playground.

Moreover, preliminary results confirm that using *codelet* queues in the scheduling engine is powerful enough to efficiently exploit the specificities of the CELL processor; and the design of OPENCL is also based on task queues: list scheduling with our `push` and `pop` operations is a simple, yet expressive paradigm.

3.3 Scheduling Hints

To investigate the possible scope of performance improvements thanks to better scheduling, we let the programmer add some extra optional scheduling hints within the *codelet* structure. One of our objective is to fill the gap between the tremendous amount of work that has been done in the field of scheduling theory and the need to benefit from those theoretical approaches on actual machines.

Declaring prioritized tasks. The first addition is to let the programmer specify the level of priority of a task. Such priorities typically prevent crucial tasks from having their execution delayed too much. While describing which tasks should be prioritized usually makes sense from an algorithmic point of view, it could also be possible to infer it provided an analysis of the task DAG.

Guiding scheduling policies with performance models. Many theoretical studies of scheduling problems often assume to have a *weighted* DAG of the tasks [2]. Whenever it is possible, we thus propose to let the programmer specify a performance model to extend the dependency graph with weights. Scheduling policies can subsequently eliminate the source of load imbalance by distributing work with respect to the amount of computation that has already been attributed to the various processing units.

The use of performance models is actually fairly common in high performance libraries. Various techniques are thus available to allow the programmer to make performance predictions. Some libraries exploit the performance models of computation kernels that have been studied extensively (*e.g.*, BLAS). This for instance makes it possible to select an appropriate granularity [17] or even a better (static) scheduling [15]. It is also possible to use sampling techniques to automatically determine such model costs, provided actual measurements. In STARPU that can be done either by the means of a pre-calibration run, using the results of previous executions, or even by dynamically adapting the model with respect to the running execution.

The heterogeneous nature of the different workers makes performance prediction even more complex. Scheduling theory literature often assumes that there is a

Table 1. Scheduling policies implemented using our interface

Policy	Category	Queue design	Load balancing
default	greedy	central FIFO	n/a
priority	greedy	central deque or priority FIFO	n/a
work-stealing	greedy	per-worker deque	steal
weighted random	directed	per-worker FIFO	model
cost model	directed	per-worker FIFO	model

mathematical model for the amount of computation (*i.e.*, in FLOP), and that execution time may be computed according to the relative speed of each processor (*i.e.*, in FLOP/S) [2]. However, it is possible that a *codelet* is implemented using different algorithms (with different algorithmic complexities) on the various architectures. As the efficiency of an algorithm heavily depends of the underlying architecture, another solution is to create performance models for each architecture. In the following section, we compare both approaches and analyze the impact of model accuracy on performances.

3.4 Predefined Scheduling Policies

We currently implemented a set of common queue designs (stack, FIFO, priority FIFO, deque) that can be directly manipulated within the different methods in a high level fashion. The policy can also decide to create different queue topologies, for instance a central queue or per-worker queues. The `push` and the `pop` methods are then responsible for implementing the load balancing strategy. Defining a policy with our model only consists in defining a couple of methods. On the one hand, a method called at the initialization of STARPU. On the other hand, the `push` and the `pop` methods that implement the interaction with the abstract queue.

Table I shows a list of scheduling policies that were designed usually in less than 100 lines of C code, which shows the conciseness of our approach.

4 Experimental Validation

To validate our approach, we present several scheduling policies and experiment them in STARPU on a few applications. To investigate the scope of improvements that scheduling can offer, we gradually increase the quality of the hints given to STARPU by the programmer. We then analyze in more details how STARPU takes advantage of proper scheduling to exploit heterogeneous machines efficiently.

4.1 Experimental Testbed

Our experiments were performed on an E5410 XEON quad-core running at 2.33 GHz with 4 GB of memory and an NVIDIA QUADRO FX4600 graphic card with 768 MB of embedded memory. This machine runs LINUX 2.6 and CUDA 2.0. We used the ATLAS 3.6 and the CUBLAS 2.0 implementations of the BLAS kernels. All measurements were performed a significant number of times and unless specified otherwise,

the standard deviation is never above 1 % of the average value which we show. Given CUDA requirements, one core is dedicated to controlling the GPU efficiently [11] so that we compute on three cores and a GPU at the same time.

We have implemented several (single precision) numerical algorithms that use *codelets* in order to analyze the behaviour of STARPU and the impact of scheduling on their performance. A **blocked matrix multiplication** which will help us demonstrate that greedy policies are not always effective, even on such simple algorithms. A **blocked Cholesky decomposition** (without pivoting) which emphasizes the need for priority-based scheduling. A **blocked LU decomposition** (without pivoting) which is similar to Cholesky but performs twice as much computation, and thus parallelism. This demonstrates how our system tackles load balancing issues while actually taking advantage of a heterogeneous platform.

All these algorithms are compute-bound as they mostly involve BLAS 3 kernels ($\mathcal{O}(n^3)$) operations against $\mathcal{O}(n^2)$ memory accesses). Performance figures are shown in synthetic GFLOP/S as this gives an evaluation of the efficiency of the computation since speedups are not relevant on such heterogeneous platforms.

4.2 Impact of the Design of the Queues

The choice of the design and the organization of the queues that compose a strategy is important when writing a scheduling strategy. The choice between a FIFO and a stack may also be important: a stack may for instance help to improve locality and thus data reuse, especially with divide-and-conquer algorithms, but implementing priority is easier with a FIFO. As all our benchmarks naturally tend to have a FIFO task ordering, measurements are not performed on stack-based strategies since that is irrelevant.

Even if it may require some load balancing mechanisms, decentralising queues helps to reduce contention and makes it possible to handle each worker specifically. This is also interesting when accessing a global shared queue is expensive (e.g., on the CELL which needs expensive DMA transfers).

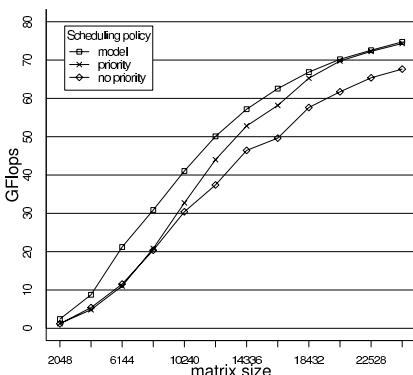


Fig. 1. Cholesky decomposition

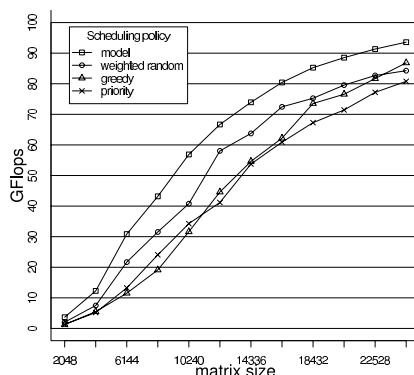


Fig. 2. LU decomposition

Scheduling policies with support for priority tasks helps reducing load imbalance for algorithms that suffer insufficient parallelism. Figure 1 shows that the Cholesky algorithm benefits from priorities by up to 10 GFlops on large problems. However, it does not affect LU decomposition a lot on Figure 2 as the FIFO ordering naturally fits the natural priorities of the algorithm. On Figure 2, priority tasks are either appended at the end of a global FIFO (*greedy* policy) or put into dedicated ones (*priority FIFO*). Small problems benefit from a strict FIFO ordering, but large ones perform better with the greedy algorithm. This confirms that selecting the best scheduling policy is not obvious. That is why we made it possible to select the scheduling policy at runtime.

4.3 Policies Based on Performance Models

Figure 3 demonstrates that the *greedy* policy delivers around 105 GFLOPS on medium-sized problems, which is about 90 % of the sum of the results achieved on a single GPU (91.3 GFLOPS) and 3 cores (25.5 GFLOPS). That relatively low efficiency is explained by the important load imbalance. Intuitively, that issue can be solved if a GPU that is n times faster than a core is given n times more tasks: next section shows how that can be achieved using performance models.

Average Acceleration-Based Performance Model. In the *weighted random* strategy, each worker is associated with a ratio that can be interpreted as an *acceleration* factor. Each time a task is submitted, a random number is generated to select one of the workers with a probability proportional to its ratio. That ratio can for instance be set by the programmer once for all on the machine, or be measured thanks to reference benchmarks (*e.g.*, BLAS kernels). The *weighted random* strategy is typically be suited to independent tasks of equal size.

Still, the shaded area on Figure 3 shows that this policy produces extremely variable schedules for which the lack of load balancing mechanism explains why the average value is worse than the *greedy* policy. Unexpectedly, this strategy also gives really interesting improvement of an order of 10 GFLOPS on LU and Cholesky decompositions

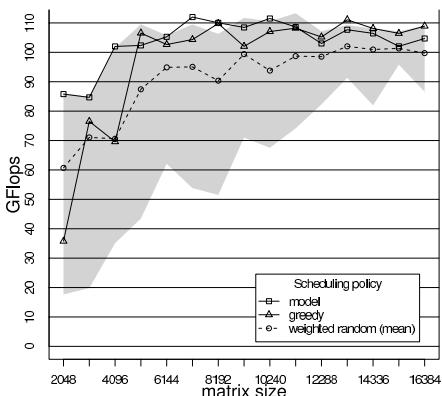


Fig. 3. Blocked Matrix Multiplication

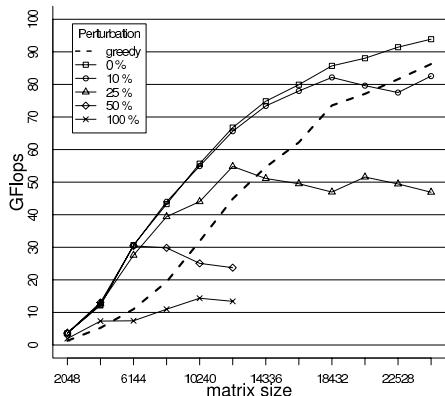


Fig. 4. Impact of the model accuracy

even though the latter is not shown on Figure 1 for readability reasons. It is interesting to note that this optimization is effective on the entire spectrum of sizes, especially on medium ones for which it is especially interesting to obtain performance improvements without any effort from programmers.

Per-Task Accurate Performance Models. Evenly distributing tasks over workers with regard to their respective speed does not necessarily make sense for tasks that are not equally expensive, or if there are dependencies between them. Hence, programmers can specify a *cost model* for each *codelet*. This model can for instance represent the amount of work and be used in conjunction with the relative speed of each workers. It can also directly model the execution time on each of the architectures. We implemented the *Earliest Task First* strategy using those models. Each worker is assigned a queue. Given their expected duration, tasks are then assigned to the queues which minimize termination time.

By severely reducing load balancing issues, we obtained substantial improvements over all our previous strategies, for all benchmarks. Even though there is a limited number of independent tasks, we always obtain almost the best execution for matrix multiplication. On the CHOLESKY decomposition, performance models outperform our other strategies on any sizes. This strategy improves the performance of medium and small sized problems up to twofold, but we observe equivalent results for large ones because the *priority* strategy does not suffer too much load imbalance on large inputs. The *performance modeling* strategy is also handicapped by a limited support for priority tasks, which is especially useful on the Cholesky benchmark. Likewise, the LU decomposition obtains even more important improvements as we achieve up to 25 GFLOPS improvement, thus reducing execution time by a factor of 2 on medium size problems. It also increases asymptotic speed from 80 GFLOPS to more than 95 GFLOPS.

On Figure 4, we applied a random perturbation of the performance prediction to study how model accuracy affects scheduling. Even with large miss-predictions (e.g., 25 %), the *performance modeling* strategy still outperforms other ones for medium size problems: the importance of accuracy depends on the size of the problem. But since the execution time is usually within less than 1 % of the prediction for BLAS kernels, it is worth paying a attention to an accurate modeling, even if it needs not give exact prediction to be useful.

4.4 Taking Advantage of Heterogeneity

In our *heterogeneous* context, we define **efficiency** as the ratio between the sum of the speeds obtained separately on each architecture and the speed obtained while using all architectures at the same time. This indeed expresses how well we manage to add up the

Table 2. Superlinear acceleration on LU decomposition (30720×30720)

	Simple performance model			Per-architecture performance model		
Measured speed (GFlops)	3 CPUs + 1 GPU 95.41	3 CPUs 21.24	1 GPU 75.04	3 CPUs + 1 GPU 98.21	3 CPUs 21.68	1 GPU 75.07
Efficiency	95.41 = 99.1 % ($21.24 + 75.04$)			98.21 = 101.5 % ($21.68 + 75.07$)		

speeds of the different architectures. Table 2 shows the efficiency of our parallel code. While heterogeneity could impact programmability, our implementation of the LU decomposition is actually not affected by the use of various architectures at the same time: the speed measured with three CPUs and a GPU amounts to 99.1 % of the sum of the speeds measured separately with three cores on the one hand, and with one GPU on the other hand. This result is contrasted by the need to dedicate one core to the accelerator, but it demonstrates that the overhead is rather low, mostly caused by parallelization rather than heterogeneity itself.

In addition to that, Table 2 shows an interesting *superlinear* efficiency of 101.5 % when using a per-architecture performance model instead of a mere *accelerating factor* with a single performance model common to all architectures. This illustrates the impossibility to model the actual capabilities of the various computation units by the means of a mere *ratio*, even though this model is fairly common in theoretical scheduling literature [2]. Some tasks indeed suit GPUs while others are relatively more efficient on CPUs: matrix multiplication may be ten times faster on a GPU than on a core while a CPU may be only five times slower on matrix additions. The rationale behind this *superlinear* efficiency is that it is better to execute the tasks you are good at, and to let others perform those for which you are not so good.

5 Related Work

If accelerators have received a lot of attention in the last years, most people program them directly on top of constructors' API at a low level, with little attention to code portability. While GPGPUs were historically programmed using standard graphical APIs [14], AMD's FIRESTREAM and especially NVIDIA's CUDA are by far the most common way to program GPUs nowadays. Likewise, CELL is usually programmed directly on top of the low level LIBSPE interface even if IBM ALF targets both CELL and multicore processors. FPGA, CLEARSPED and all other accelerating boards still need specific vendor interfaces. Most efforts however tend to be around writing fast computation kernels rather than designing a generic programming model.

In contrast, multicore (and SMP) programming is getting more mature and *standards* such as OPENMP, which has gained substantial audience in spite of MPI which still remains the most commonly used standard in HPC. Besides the OPENCL standardization effort which not only attempts to unify programming paradigms, but also proposes a low-level device interface, a lot of projects thus try to implement the MPI standard on the CELL [13] while it does not seem to be adapted for GPUs even if it becomes common to use hybrid models (*e.g.*, CUDA with MPI processes). DURAN *et al.* propose to enrich OPENMP with directives to declare data dependencies [8], which is particularly useful for all accelerator technologies. OPENMP therefore seems to be a promising programming interface for both CELL [4] and GPUs provided compiling environments are offered sufficient support. STARPU could thus be used as a back-end for CELLSS or for the HMPP [7] which generate *codelets* resp. for the CELL and for GPUs.

A lot of efforts have also been devoted to design or to extend languages with a proper support for data and task parallelism, but most of them actually re-implement a streaming paradigm [11] which does not necessarily capture all applications that may exploit accelerators. Various projects intend to implement libraries with computation kernels

that are actually offloaded [5], but STARPU avoids for instance the limitation of the size of problems solved by BARRACHINA *et al.* [3] while preserving the benefits of their work at the algorithmic level.

Some runtime systems were designed to address multicore and accelerators architectures [6][12][16]. Most approaches adopt an interface similar to CHARM++’s asynchronous OFFLOAD API. The well established CHARM++ runtime system actually offers support for both CELL [10] and GPUs [16] (even though there are no performance evaluation available yet for GPUs to the best of our knowledge). But its rather low level interface only has limited support for data management: offloaded tasks only access blocks of data instead of our high level arbitrary data structures, and they do not benefit from our caching techniques. JIMENEZ *et al.* use performance prediction to schedule tasks between a CPU and a GPU [9], but their approach is not applicable to scheduling inter-dependent tasks since data transfers are explicit.

6 Conclusion and Future Work

We presented STARPU, a new runtime system that efficiently exploits heterogeneous multicore architectures. It provides a uniform execution model, a high-level framework to design scheduling policies and a library that automates data transfers. We have written several scheduling strategies and observed how they perform on some classical numerical algebra problems.

In addition to improving programmability by the means of a high level uniform approach, we have shown that applying simple scheduling strategies can significantly reduce load balancing issues and improve data locality. Since there exists no ultimate scheduling strategy that addresses all algorithms, programmers who need to hard-code task scheduling within their hand-tuned code may experiment important difficulties to select the most appropriate strategy. Many parameters may indeed influence which policy is best suited for a given input. Empirically selecting at runtime the most efficient one makes it possible to benefit from scheduling without putting restrictions or making excessive assumptions. We also demonstrated that given a proper scheduling, it is possible to exploit the specificity of the various computation units of a heterogeneous platform and to obtain a consistent superlinear efficiency.

It is crucial to offer a uniform abstraction of the numerous programming interfaces that result from the advent of accelerator technologies. Unless such a common approach is adopted, it is very unlikely that accelerators will evolve from a *niche* with dispersed efforts to an actual mainstream technique. While the OPENCL standard also does provide task queues and an API to offload tasks, our work shows that such a programming model needs to offer an interface that is simple but also expressive.

We plan to implement our model on additional accelerator architectures, such as the CELL by the means of a driver for the CELL RUNTIME LIBRARY [12], or on top of generic accelerators with an OPENCL driver. In the future, we expect STARPU to offer support for the HMPP compiling environment [7] which could generate our *codelets*. We are also porting real applications such as the PASTIX [15] and the MUMPS solvers. STARPU could be a high-level platform to implement some of the numerous policies that exist in the scheduling literature. This would reduce the gap between HPC applications and theoretical works in the field of scheduling.

References

1. Augonnet, C., Namyst, R.: A unified runtime system for heterogeneous multicore architectures. In: Euro-Par 2008 Workshops - Parallel Processing, Las Palmas de Gran Canaria, Spain (August 2008)
2. Banino, C., Beaumont, O., Carter, L., Ferrante, J., Legrand, A., Robert, Y.: Scheduling strategies for master-slave tasking on heterogeneous processor platforms. *IEEE Trans. Parallel Distrib. Syst.* 15(4), 319–330 (2004)
3. Barrachina, S., Castillo, M., Igual, F.D., Mayo, R., Quintana-Ort, E.S.: Solving Dense Linear Systems on Graphics Processors. Technical report, Universidad Jaime I, Spain (February 2008)
4. Bellens, P., Perez, J.M., Badia, R.M., Labarta, J.: Cellss: a programming model for the cell be architecture. In: SC 2006: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, p. 86. ACM, New York (2006)
5. Buttari, A., Langou, J., Kurzak, J., Dongarra, J.: A class of parallel tiled linear algebra algorithms for multicore architectures (2007)
6. Crawford, C.H., Henning, P., Kistler, M., Wright, C.: Accelerating computing with the cell broadband engine processor. In: CF 2008 (2008)
7. Dolbeau, R., Bihan, S., Bodin, F.: HMPP: A hybrid multi-core parallel programming environment (2007)
8. Duran, A., Perez, J.M., Ayguade, E., Badia, R., Labarta, J.: Extending the openmp tasking model to allow dependant tasks. In: IWOMP Proceedings (2008)
9. Jiménez, V.J., Vilanova, L., Gelado, I., Gil, M., Fursin, G., Navarro, N.: Predictive runtime code scheduling for heterogeneous architectures. In: HiPEAC, pp. 19–33 (2009)
10. Kunzman, D.: Charm++ on the Cell Processor. Master's thesis, Dept. of Computer Science, University of Illinois (2006)
11. McCool, M.D.: Data-parallel programming on the cell be and the gpu using the rapidmind development platform. In: GSPx Multicore Applications Conference (2006)
12. Nijhuis, M., Bos, H., Bal, H.E., Augonnet, C.: Mapping and synchronizing streaming applications on cell processors. In: HiPEAC, pp. 216–230 (2009)
13. Ohara, M., Inoue, H., Sohda, Y., Komatsu, H., Nakatani, T.: Mpi microtask for programming the cell broadband enginetm processor. *IBM Syst. J.* 45(1) (2006)
14. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A.E., Purcell, T.J.: A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum* 26(1), 80–113 (2007)
15. Ramet, P., Roman, J.: Pastix: A parallel sparse direct solver based on a static scheduling for mixed 1d/2d block distributions. In: Proceedings of Irregular'2000, Cancun, Mexique, pp. 519–525. Springer, Heidelberg (2000)
16. Wesolowski, L.: An application programming interface for general purpose graphics processing units in an asynchronous runtime system. Master's thesis, Dept. of Computer Science, University of Illinois (2008)
17. Whaley, R.C., Dongarra, J.: Automatically Tuned Linear Algebra Software. In: Ninth SIAM Conference on Parallel Processing for Scientific Computing (1999)

XJava: Exploiting Parallelism with Object-Oriented Stream Programming

Frank Otto, Victor Pankratius, and Walter F. Tichy

University of Karlsruhe, 76131 Karlsruhe, Germany
`{otto,pankratius,tichy}@ipd.uka.de`

Abstract. This paper presents the XJava compiler for parallel programs. It exploits parallelism based on an object-oriented stream programming paradigm. XJava extends Java with new parallel constructs that do not expose programmers to low-level details of parallel programming on shared memory machines. Tasks define composable parallel activities, and new operators allow an easier expression of parallel patterns, such as pipelines, divide and conquer, or master/worker. We also present an automatic run-time mechanism that extends our previous work to automatically map tasks and parallel statements to threads.

We conducted several case studies with an open source desktop search application and a suite of benchmark programs. The results show that XJava reduces the opportunities to introduce synchronization errors. Compared to threaded Java, the amount of code could be reduced by up to 39%. The run-time mechanism helped reduce effort for performance tuning and achieved speedups up to 31.5 on an eight core machine.

1 Introduction

With multicore chips, software engineers are challenged to provide better performance by exploiting parallelism. Although well-established languages such as C++ or Java use the thread model for parallel execution, this model has turned out to be error-prone and difficult to handle for large programs. Programmers are forced to think on low abstraction levels and consider many details of synchronization. As a result, synchronization bugs, deadlocks, or data races are likely to occur. The growing complexity of multithreaded software makes it painful to locate such defects.

Several of our earlier case studies on parallel programming in various areas [10,11] show that best performance was achieved with a structured approach, considering different abstraction layers and patterns for parallelism. However, despite clear objectives and reasonable designs of the parallel programs, the implementation has always required a lot of effort.

XJava [9] is a programming language that extends Java with language constructs for parallelism, combining concepts from object-orientation and stream languages. Parallelism is exploited without requiring the programmer to explicitly define threads. The central construct in XJava is the task, which is syntactically similar to a method and which encapsulates an activity to be executed in

parallel. Tasks can be combined with special operators to obtain parallel statements that express master/worker or pipeline parallelism.

The XJava compiler basically performs a source-to-source transformation; XJava code is translated into native, instrumented Java code which is eventually translated into bytecode. The XJava compiler divides tasks and parallel statements into logical code units that are passed to a built-in scheduler. The scheduler transparently employs a fixed number of dedicated executor threads to execute these units. This approach intends to reduce the degree of non-determinism in execution behavior. The scheduling algorithms can easily be replaced or extended in the future to support real-time load balancing.

Our case studies show that XJava makes parallel programming simpler and more productive; there is less potential of inadvertently introducing synchronization bugs. To evaluate XJava’s capabilities, we considered a desktop search engine and a selection of smaller benchmark programs. For each application, XJava code was compared in detail with equivalent threaded code as well as sequential code. Compared to threaded Java, XJava lead to code savings of up to 39% and required 87.5% – 100% less manual synchronization. We achieved speedups between 1.5 and 31.5 over sequential Java on an eight core machine.

In this paper, we briefly sketch the XJava extensions introduced in our previous work [9] and present the key details of the mechanisms in the compiler and runtime system. In addition, we present new results that demonstrate XJava’s ability to implement parallel design patterns on several abstraction levels.

2 The XJava Language

XJava [9] extends Java with more intuitive constructs for general-purpose parallel programming. Several types of parallelism can be expressed in this language, although we only introduce two new keywords and two new operators.

XJava combines the principles of object-orientation with stream languages. XJava’s design intends to make programs behave as programmer would intuitively expect. Thus, code understanding and maintainability should become easier as well. XJava makes parallelism accessible at higher abstraction levels and hides manual synchronization.

2.1 Tasks

Tasks are key constructs and are in fact special methods. Tasks encapsulate parallel activities that run within a thread, and can be combined to *parallel statements* (cf. Section 2.2). In contrast to methods, tasks have no return type, but typed input and output ports. A public task with input type A and output type B is declared as

```
public A => B t() { ... }
```

The input port receives a stream of data with elements of type A, and a stream of data elements of type B is sent to the output port. If a task has no input or

output, the respective port type is `void`. We distinguish between two kinds of tasks: *periodic tasks* and *non-periodic tasks*.

Periodic tasks define exactly one `work` block that is repeatedly executed for each stream element. For example, the code

```
String => String encode(Key key) {
    work(String s) { push encrypt(s, key); }
}
```

declares a periodic task that expects an input stream of `String` elements. The current element of the input stream is assigned to the local variable declared in the parenthesis after the `work` keyword. The work block – in this case, the encryption routine – is repeatedly executed for each element of the input stream. Each received element is encrypted with the key passed as an argument to the task. The `push` statement appends the encrypted element, i.e., a `String` object, to the output stream. The work block terminates when an end-of-stream token is received on the input port.

By contrast, **non-periodic tasks** do not have a work block, i.e., their code is executed only once. The body of a non-periodic task may contain a parallel statement for introducing nested parallelism.

2.2 Parallel Statements

Parallel statements are used to combine tasks to more complex parallel constructs. They consist of tasks that are concatenated by the operators “`=>`” and “`|||`”.

The “`=>`” operator creates a *pipe statement*; it connects tasks on their input and output ports. This operator can be used to build pipelines of arbitrary lengths. For example, consider the tasks

```
void => String read(File fin) { /* ... */ }
String => void write(File fout) { /* ... */ }
```

`read` reads some file and turns its content into a stream of `String` objects; `write` accepts a stream of `String` objects and writes them to an output file. For given input and output files `fin` and and some key `key`, the statement

```
read(fin) => encode(key) => write(fout);
```

creates a pipeline for encoding a file and automatically exploits pipeline parallelism.

Concurrent statements are defined by the “`|||`” operator that concurrently executes tasks that are not allowed to be connected neither input nor output ports. For example, consider a task for simulating work based on randomly generated events:

```
void => void simulateW() { /* ... */ }
```

Suppose we want to run several simulations to collect data or retrieve statistical average values, we can use the following statements:

```
simulateW() ||| simulateW() ||| simulateW();    // 3 tasks
simluateW():i;                                // i tasks
```

The first statement concurrently executes `simulateW` three times. Alternatively, the “`:`” operator can be used to define the number of parallel tasks dynamically.

3 The XJava Compiler

The XJava compiler extends the Polyglot compiler framework [8]. The XJava compiler checks if task declarations and parallel statements are valid and produces Java code that is instrumented with calls to the XJava scheduler (cf. Section 4).

3.1 Compiling Periodic Tasks

A periodic task defines exactly one `work` block; an arbitrary number of Java statements may precede or follow this block. Thus, the body of a periodic task can be divided into three parts that we call *BW* (“before work”), *W* (“work”) and *AW* (“after work”). Consider a slightly modified form of the `encode` task from Section 2.1 with additional code before and after the work block:

```
public String => String encode(Key) {
    ... /* Java code */                                /* BW */
    work (String s) { push encrypt(s, key); } /* W */
    ... /* Java code */                                /* AW */
}
```

This task declaration is compiled to a wrapper class `EncodeWrapper`. Its superclass `PeriodicTask` is provided by the XJava runtime library; it is an abstract class defining the methods `beforeWork()`, `work(int)` and `afterWork()`. These methods are implemented in the wrapper class and contain the Java code before, in, and after the work block. The purpose of this separation is to divide a task into logical units that can be executed individually by the scheduler. This reduces the degree of non-determinism makes parallel execution easier to predict.

```
class EncodeWrapper extends PeriodicTask {
    Connector cin, cout;
    ...
    beforeWork() { ... /* Java code for BW */ }
    work(int n) { /* repeatedly called by the scheduler */
        for (int i = 0; i < n; i++) {
            ... /* Java code for W */
        }
    }
    afterWork() { ... /* Java code for AW */ }
}
```

Connector is a class for buffering elements that are exchanged between tasks and is part of XJava's runtime framework. **cin** and **cout** each contain buffers to receive or send elements. Whenever **encrypt** expects a new incoming element in the previous example, it calls the respective method of the buffer in **cin**; a **push** statement is mapped to a call to a similar method of the **cout** buffer. **work(int n)** provides a method to the scheduler to execute **n** iterations of the work block (cf. Section 4). The number **n** of iterations is determined by the scheduler. Local variables of each task are saved, i.e., each task instance has an internal state. When **work** has no more iterations to do and terminates, the **afterWork** method is called. In addition, the task will close its output stream and will be marked as finished.

3.2 Compiling Non-periodic Tasks

Since non-periodic tasks do not define a work block, their bodies cannot be separated. A non-periodic task **foo** is compiled to a wrapper class extending the class **NonPeriodicTask** in XJava's runtime framework. This class implements Java's **Runnable** interface; the **run** method contains Java code representing the body of the task:

```
class FooWrapper extends NonPeriodicTask {
    Connector cin, cout;
    ...
    run() { /* Java code for foo's body */ }
}
```

3.3 Compiling Parallel Statements

Task calls and parallel statements are managed by XJava's scheduler. Whenever a task is called, an instance of its corresponding class will be created and passed on to the scheduler. The scheduler decides when and how to execute it (cf. Section 4).

Parallel statements consist of a number of tasks connected by operators. The compiler checks in a parallel statement if input and output types of tasks match. Each task's input and output connectors are selected according to the operators involved. For example, consider the previous example of a pipe statement for encoding a file:

```
read(fin) => encode(key) => write(fout);
```

This statement is compiled to

```
Connector c1 = new Connector(); Connector c2 = new Connector();
ReadWrapper r = new ReadWrapper(fin, c1);
EncodeWrapper c = new EncodeWrapper(f, c1, c2);
WriteWrapper w = new WriteWrapper(fout, c2);
Scheduler.add(r); Scheduler.add(c); Scheduler.add(w);
w.join();
```

For each wrapper class, **connectors** as well as the arguments of the corresponding task call are passed to the constructor. The task calls themselves are mapped to calls of `Scheduler.add(Task)`. After each parallel statement, the compiler injects a join call; this method is part of XJava's runtime library and ensure that statements following the parallel statement will only execute after all tasks are finished. For a pipe statement, only the last task of the statement needs to be joined as it also finishes last. For a concurrent statement, there is no time order, so each task of that statement needs to be joined.

4 Scheduling Mechanism

Section 3 described how task declarations, task calls and parallel statements are compiled. Figure 1 shows how tasks are passed to and executed by the scheduler. The scheduler was designed to avoid unbounded growth of thread numbers and provide efficient execution. Both periodic and non-periodic tasks are compiled to wrapper classes; task calls are mapped to new instances of these classes and passed on to the scheduler.

When a **periodic task** PT is called, its instance is added to the task queue. The scheduler employs a fixed number of executor threads that take task instances from the queue, execute some iterations of their work method, and enqueue the task instance back. By default, the number of executor threads is equal to the number of CPU cores that are available. The number of work iterations is fixed and a default value is globally defined. In the future, we plan to dynamically adjust this value based on the execution behavior and work load. Tasks that are currently waiting for input elements are moved to the end of the queue. This mechanism prioritizes tasks for execution depending on the amount of work to be done.

Non-periodic tasks are also passed on to the scheduler, but they are not suitable for partial execution by executor threads. Instead, the scheduler decides based on the number of currently active threads and available memory, whether

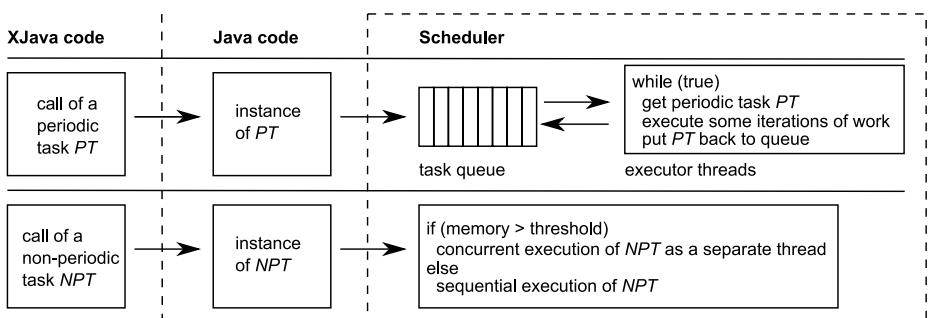


Fig. 1. Periodic and non-periodic tasks are compiled and passed on to XJava's scheduler. The scheduler provides executor threads for periodic tasks. Non-periodic tasks are executed either concurrently in separate threads or sequentially.

to execute a particular task sequentially by directly calling its run method, or to execute it in a separate thread.

5 Implementing Parallel Design Patterns

Parallel design patterns as described by Mattson et al. [7] provide useful principles for developing parallel software. However, the gap between patterns and program code has to be bridged by the programmer. XJava's language constructs help bridge that gap; the input-output-mechanism of tasks and their compositability allow for expressing several types of parallelism such as pipelines, divide and conquer parallelism, or master/worker. As pipelines were already illustrated earlier, we now provide examples for divide and conquer and master/worker parallelism.

5.1 Divide and Conquer Algorithms

Divide and conquer algorithms are easy to express in XJava, as the parallel code is similar to the sequential version. For example, the following code sketches a sequential merge sort algorithm:

```
void mergesort(int from, int to) {
    if (to - from > threshold) {
        int x = from + (to - from)/2;
        mergesort(from, x);
        mergesort(x + 1, to);
        merge(from, x + 1, to);
    } else sort(from, to);
}
```

Parallelizing this algorithm in XJava is simple. (1) We need to make the method `mergesort` a `void => void` task. (2) Instead of sequentially calling `mergesort` two times in the recursion step, we combine these calls to a concurrent statement using the “`|||`” operator:

```
void => void mergesort(int from, int to) { // (1)
    if (to - from > threshold) {
        int x = from + (to - from)/2;
        mergesort(from, x) ||| mergesort(x + 1, to); // (2)
        merge(from, x + 1, to);
    } else sort(from, to);
}
```

The programmer does not need to think about synchronization or the number of running threads. This example shows how efficiently divide-and-conquer-based parallelism can be expressed in XJava. More details and performance results of this algorithm are given in Section [6].

5.2 Master/Worker Configurations

The master/worker pattern is a powerful concept that is used in many parallel applications. In XJava, it can be created by parallel statements that use variants of the “=>” operator. For example, consider a file indexing algorithm. The task declaration

```
File => void index() { /* ... */ }
```

specifies a worker task expecting a stream of `File` objects in order to index them. The non-periodic task

```
File => void indexers(int i) {
    index():i;
}
```

encapsulates a set of `i` workers that run concurrently to make indexing parallel. The task

```
void => File visitFiles(File root) { /* ... */ }
```

can be interpreted as a master task that recursively visits all files in the directory specified by the `root` argument. For a `File` object `dir` that we want to index, we can easily create a master/worker configuration employing a certain number, say `n`, workers:

```
visitFiles(dir) => indexers(n);
```

By choosing the “=>” operator, the workers are fed in a round-robin style with elements from the stream. Alternatively, the “=>?” operator distributes elements on a first-come-first-serve basis; the “=>*” operator would broadcast each element to all workers. In the file indexing context, the “=>?” apparently makes most sense: the order in which files are indexed is not important. In addition, the worker’s waiting times are reduced, which results in better performance.

6 Experimental Results

We evaluated the sequential (i.e., single-threaded), multi-threaded, and XJava versions of four benchmark programs to compare code structure and performance. The threaded versions were written before the XJava versions to make sure that programs do not just re-implement the XJava concepts. We chose the programs to cover data, task, and pipeline parallelism. All programs were tested on three different machines: (1) an Intel Quadcore Q6600 with 2.40 GHz, 8 GB RAM and Windows XP Professional; (2) two Intel Xeon Quadcores E5320 with 1.86 GHz (Dual-Quadcore), 8 GB RAM and Ubuntu Linux 7.10; (3) a Sun Niagara 2 with 8 cores at 1.2 GHz capable of executing 4 threads, 16 GB RAM, and Solaris 10.

6.1 The Benchmark

JDesktopSearch (JDS) [5] is an open source desktop search application written entirely in Java. We use it as an example of a realistic, major application. It consists of about 3,400 lines of code (LOC) plus several additional libraries. The file indexing part is already multithreaded, but it can also be executed with just one thread. We re-engineered this part from Java to XJava; the relevant code had 394 LOC. We split the indexing methods into tasks to implement a master/worker approach as described in Section 5.2. Conceptually, the master task recursively walks through the root directory that has to be indexed, pushing the contained indexable files to the workers. We tested JDS for a 242 MB sample directory containing different file types and file sizes with a total of 12,223 files in 3,567 folders.

Additionally, we considered a selection of three smaller programs that are standard examples for parallelization. *Mandelbrot* computes mandelbrot sets based on a given resolution and a maximum number of iterations. *Matrix* multiplies matrices of type `double` that are randomly generated. *MergeSort* sorts an array of 3 million randomly generated integer values. It is a representative of divide and conquer algorithms; the XJava version implements the code already sketched in Section 5.1.

6.2 Results

Code. Table 2 shows code metrics for the sequential, threaded and XJava versions of the benchmark programs. In addition, the table presents total and relative improvements achieved by XJava over threaded Java. Outstanding benefits of XJava over threaded Java are that XJava saves code and significantly reduces the need for manual synchronization. For MergeSort, the sizes of the sequential and XJava programs are the same; and compared to the threaded MergeSort, XJava saved 39% of code. In XJava, fewer classes are required; the number of attributes is lower since attributes for synchronization or status notifications are no longer needed. We counted the occurrences of `synchronized`, `wait`, `join`, `notify`, `notifyAll`, `sleep`, and `java.util.concurrent`. For the desktop search application, the number of synchronization constructs was reduced from 8 to 1; the other XJava benchmark programs do not require any manual synchronization at all. This effect comes from XJava’s high abstraction level and its implicit synchronization mechanisms. Accordingly, the number of try-catch blocks is reduced for all programs since exceptions caused by concurrent modifications or interrupted threads do not need to be considered anymore. Our XJava programs used a few more methods (including tasks) than threaded Java, which lead to marginally lower average nested block depths.

Performance. We measured the performance of all benchmark programs’ sequential, threaded and XJava versions; results are shown in Figure 3. Overall, XJava can indeed compete with threaded Java as execution times are on the same level. Only for divide and conquer parallelism as in the MergeSort program,

		JDS	%	Mandelbrot	%	Matrix	%	Mergesort	%
LOC	seq			86		51		72	
	par	394		116		90		118	
	xjava	362		99		61		72	
	improvement	32	8,1%	17	14,7%	29	32,2%	46	39,0%
classes	seq			1		1		1	
	par	2		2		3		2	
	xjava	1		2		1		1	
	improvement	1	50,0%	0	0,0%	2	66,7%	1	50,0%
attributes	seq			6		4		2	
	par	11		13		7		4	
	xjava	8		13		4		2	
	improvement	3	27,3%	0	0,0%	3	42,9%	2	50,0%
methods	seq			5		2		4	
	par	28		6		3		6	
	xjava (methods)	26		4		1		3	
	xjava (tasks)	3		3		3		1	
	improvement	-1	-3,6%	-1	-16,7%	-1	-33,3%	2	33,3%
synchronization*	seq			0		0		0	
	par	8		1		1		1	
	xjava	1		0		0		0	
	improvement	7	87,5%	1	100,0%	1	100,0%	1	100,0%
exceptions (try-catch, throw)	seq			1		0		0	
	par	34		3		3		1	
	xjava	32		1		0		0	
	improvement	2	5,9%	2	66,7%	3	100,0%	1	100,0%
nested block depth (average)	seq			2,33		2,75		2,20	
	par	1,85		2,43		3,00		2,38	
	xjava	1,84		2,00		2,17		2,20	
	improvement	0,01	0,5%	0,43	17,7%	0,83	27,8%	0,18	7,4%
nested block depth (standard deviation)	seq			0,94		1,09		0,75	
	par	1,48		0,90		1,27		0,99	
	xjava	1,47		1,12		0,90		0,75	
	improvement	0,00	0,3%	-0,21	-23,7%	0,37	29,0%	0,24	24,6%
nested block depth (maximum)	seq			3,00		4,00		3,00	
	par	8,00		3,00		5,00		4,00	
	xjava	8,00		4,00		3,00		3,00	
	improvement	0,00	0,0%	-1,00	-33,3%	2,00	40,0%	1,00	25,0%

* occurrences of Strings "synchronized", "wait", "join", "notify"/"notifyAll", "sleep", "java.util.concurrent"

Fig. 2. Code metrics of the benchmark programs. In terms of code sizes and used synchronization constructs, XJava shows significant improvements over threaded Java.

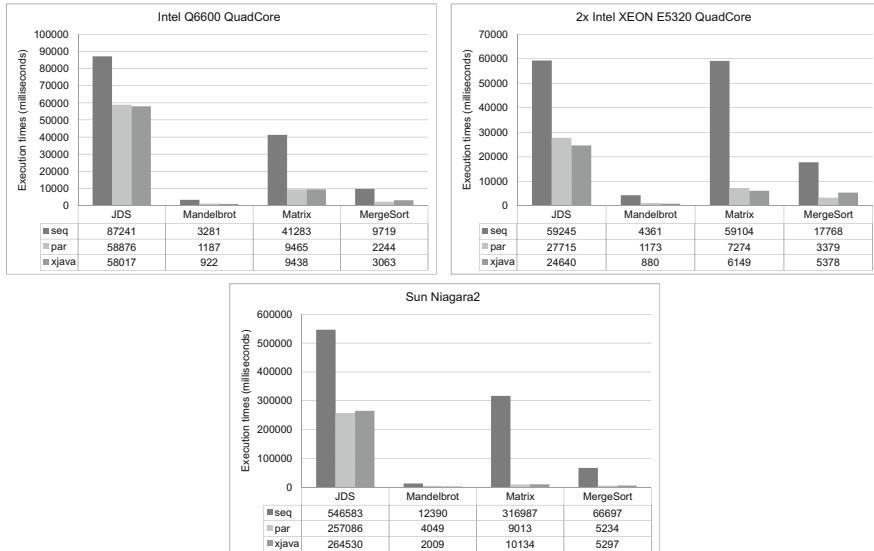


Fig. 3. Performance results of the sequential (seq), threaded (par) and XJava (xjava) versions of the benchmark programs

XJava tends to be a bit slower on all machines; on the Dual-Quadcore machine, the threaded version is even 1.6 times faster than the XJava version. The reason is most likely some overhead in the scheduler when executing non-periodic tasks. The Matrix program achieved an almost linear speedup on all machines, both for threaded Java and XJava. The speedups of the desktop search application were similar for Java and XJava; the maximum of only 2.1 can be explained with the memory bandwidth and bus bottlenecks.

Summary. Compared to threaded Java, the benchmark programs show that XJava simplifies parallel programming by reducing the amount of code, especially the need for manual synchronization and exception handling. We achieved speedups that can compete with the performance of threaded Java, although there is still potential for optimizing the scheduling mechanism.

7 Related Work

XJava is inspired by stream-oriented languages [14]. Stream programs consist of filters that are connected to form a stream graph. An input data stream of arbitrary length flows through that graph and is processed by the filters. Stream languages such as StreamIt have been demonstrated to efficiently exploit data, task, and pipeline parallelism for applications from the signal processing and graphics domain [15,4]. These languages were not designed for programming general-purpose applications; they use rather simple data structures and do not support object-orientation.

Chapel [2], Fortress [12] and X10 [3] are object-oriented languages for parallel programming. However, all of them are designed for explicit multithreading and require considerable manual synchronization. Their programming models focus on aspects of data and task parallelism; streams, master/slave or pipeline parallelism are not addressed explicitly.

As an alternative to new languages or language extensions, several libraries were designed to simplify parallel programming. Those libraries usually provide thread pools, data structures such as futures, locking mechanisms and synchronization constructs such as barriers. Intel's Threading Building Blocks [13] and Boost [11] are C++ libraries; the `java.util.concurrent` package [6] offers classes and interfaces for concurrent programming in Java. In contrast to native language constructs, the semantic information of library constructs is less powerful to enable more advanced optimizations or debugging – both essential in parallel programming.

8 Conclusion

XJava extends Java by providing tasks as native language constructs, which can be combined to parallel statements. In addition, XJava simplifies the implementation of parallel programming patterns and moves a significant part of low-level synchronization behind the scenes. Focusing in this paper on XJava's compiler

and scheduling mechanism, we benchmarked four XJava programs on three different multicore machines. Each XJava program was compared to equivalent threaded and sequential versions in terms of performance and code structure. XJava's approach is indeed applicable on programs of different complexity. We achieved speedups between 1.5 and 31.5 over sequential Java and, compared to threaded Java, code savings up to 39%.

Future work will include more case studies and experiments with applications from different domains. Also, more research is needed to evaluate different scheduling strategies in the context of XJava. A special focus will be on real-time tuning of scheduling parameters.

Acknowledgments. We thank the University of Karlsruhe and the Excellence Initiative for their support.

References

1. Boost C++ Libraries, <http://www.boost.org/>
2. Chamberlain, B.L., Callahan, D., Zima, H.P.: Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.* 21(3) (August 2007)
3. Charles, P., et al.: X10: an object-oriented approach to non-uniform cluster computing. In: Proc. OOPSLA 2005. ACM Press, New York (2005)
4. Gordon, M.I., Thies, W., Amarasinghe, S.: Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In: Proc. ASPLOS-XII. ACM Press, New York (2006)
5. JDesktopSearch, <http://sourceforge.net/projects/jdesktopsearch>
6. Lea, D.: The java.util.concurrent synchronizer framework. *Sci. Comput. Program.* 58(3) (2005)
7. Mattson, T.G., Sanders, B.A., Massingill, B.L.: Patterns for parallel programming. Addison-Wesley, Boston (2005)
8. Nystrom, N., Clarkson, M.R., Myers, A.C.: Polyglot: An extensible compiler framework for java. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 138–152. Springer, Heidelberg (2003)
9. Otto, F., Pankratius, V., Tichy, W.F.: High-level Multicore Programming with XJava. In: ICSE 2009, New Ideas And Emerging Results. ACM Press, New York (2009)
10. Pankratius, V., Jannesari, A., Tichy, W.F.: Parallelizing BZip2. A Case Study in Multicore Software Engineering. Accepted for IEEE Software (September 2008)
11. Pankratius, V., Schaefer, C., Jannesari, A., Tichy, W.F.: Software engineering for multicore systems: an experience report. In: Proc. IWMSE 2008. ACM Press, New York (2008)
12. Project Fortress, <http://projectfortress.sun.com/>
13. Reinders, J.: Intel Threading Building Blocks. O'Reilly Media, Inc., Sebastopol (2007)
14. Stephens, R.: A Survey of Stream Processing. *Acta Informatica* 34(7) (1997)
15. Thies, W., Karczmarek, M., Amarasinghe, S.: StreamIt: A language for streaming applications. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, p. 179. Springer, Heidelberg (2002)

JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA

Yonghong Yan, Max Grossman, and Vivek Sarkar

Department of Computer Science, Rice University
`{yanyh,jmg3,vsarkar}@rice.edu`

Abstract. A recent trend in mainstream desktop systems is the use of general-purpose graphics processor units (GPGPUs) to obtain order-of-magnitude performance improvements. CUDA has emerged as a popular programming model for GPGPUs for use by C/C++ programmers. Given the widespread use of modern object-oriented languages with managed runtimes like Java and C#, it is natural to explore how CUDA-like capabilities can be made accessible to those programmers as well. In this paper, we present a programming interface called JCUDA that can be used by Java programmers to invoke CUDA kernels. Using this interface, programmers can write Java codes that directly call CUDA kernels, and delegate the responsibility of generating the Java-CUDA bridge codes and host-device data transfer calls to the compiler. Our preliminary performance results show that this interface can deliver significant performance improvements to Java programmers. For future work, we plan to use the JCUDA interface as a target language for supporting higher level parallel programming languages like X10 and Habanero-Java.

1 Introduction

The computer industry is at a major inflection point in its hardware roadmap due to the end of a decades-long trend of exponentially increasing clock frequencies. It is widely agreed that spatial parallelism in the form of multiple homogeneous and heterogeneous power-efficient cores must be exploited to compensate for this lack of frequency scaling. Unlike previous generations of hardware evolution, this shift towards multicore and manycore computing will have a profound impact on software. These software challenges are further compounded by the need to enable parallelism in workloads and application domains that have traditionally not had to worry about multiprocessor parallelism in the past. Many such applications are written in modern object-oriented languages like Java and C#.

A recent trend in mainstream desktop systems is the use of general-purpose graphics processor units (GPGPUs) to obtain order-of-magnitude performance improvements. As an example, NVIDIA’s Compute Unified Device Architecture (CUDA) has emerged as a popular programming model for GPGPUs for use by C/C++ programmers [1]. Given the widespread use of managed-runtime execution environments, such as the Java Virtual Machine (JVM) and .Net platforms, it is natural to explore how CUDA-like capabilities can be made accessible to programmers who use those environments.

In this paper, we present a programming interface called JCUDA that can be used by Java programmers to invoke CUDA kernels. Using this interface, programmers can write Java codes that directly call CUDA kernels without having to worry about the details of bridging the Java runtime and CUDA runtime. The JCUDA implementation handles data transfers of primitives and multidimensional arrays of primitives between the host and device. Our preliminary performance results obtained on four double-precision floating-point Java Grande benchmarks show that this interface can deliver significant performance improvements to Java programmers. The results for Size C (the largest data size) show speedups ranging from $7.70\times$ to $120.32\times$ with the use of one GPGPU, relative to CPU execution on a single thread.

The rest of the paper is organized as follows. Section 2 briefly summarizes past work on high performance computing in Java, as well as the CUDA programming model. Section 3 introduces the JCUDA programming interface and describes its current implementation. Section 4 presents performance results obtained for JCUDA on four Java Grande benchmarks. Finally, Section 5 discusses related work and Section 6 contains our conclusions.

2 Background

2.1 Java for High Performance and Numerical Computing

A major thrust in enabling Java for high performance computing came from the Java Grande Forum (JGF) [2], a community initiative to promote the use of the Java platform for compute-intensive numerical applications. Past work in the JGF focused on two areas: Numerics, which concentrated on issues with using Java on a single CPU, such as complex arithmetic and multidimensional arrays, and Concurrency, which focused on using Java for parallel and distributed computing. The JGF effort also included the development of benchmarks for measuring and comparing different Java execution environments, such as the JGF [34] and SciMark [5] benchmark suites.

The Java Native Interface (JNI) [6], Java's foreign function interface for executing native C code, also played a major role in JGF projects, such as enabling Message Passing Interface (MPI) for Java [7]. In JNI, the programmer declares selected C functions as `native` external methods that can be invoked by a Java program. The native functions are assumed to have been separately compiled into host-specific binary code. After compiling the Java source files, the `javah` utility can be used to generate C header files that contain stub interfaces for the native code. JNI also supports a rich variety of callback functions to enable native code to access Java objects and services.

2.2 GPU Architecture and the CUDA Programming Model

Driven by the insatiable demand for realtime, high-definition 3D gaming and multimedia experiences, the programmable GPU (Graphics Processing Unit) has

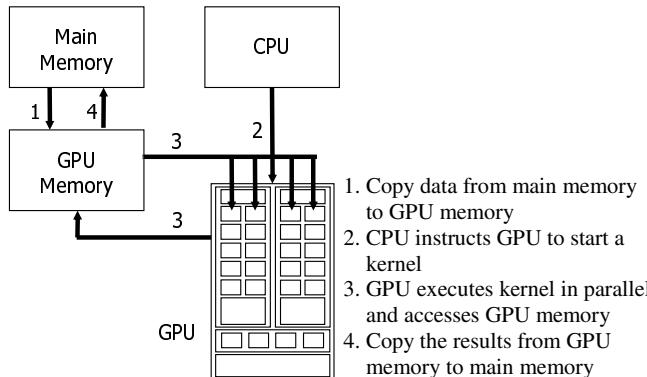


Fig. 1. Process Flow of a CUDA Kernel Call

evolved into a highly parallel, multithreaded, manycore processor. Current GPUs have tens or hundreds of fragment processors, and higher memory bandwidths than regular CPUs. For example, the NVIDIA GeForce GTX 295 graphics card comes with two GPUs, each with 240 processor cores, and 1.8 GB memory with 233.8 GB/s bandwidth, which is about 10 \times faster than that of current CPUs. The same GPU is part of the NVIDIA Tesla C1060 Computer Processor, which is the GPU processor used in our performance evaluations.

The idea behind general-purpose computing on graphics processing units (GPGPU) is to use GPUs to accelerate selected computations in applications that are traditionally handled by CPUs. To overcome known limitations and difficulties in using graphics APIs for general-purpose computing, GPU vendors and researchers have developed new programming models, such as NVIDIA's Compute Unified Device Architecture (CUDA) model [1], AMD's Brook+ streaming model [8], and Khronos Group's OpenCL framework [9].

The CUDA programming model is an extension of the C language. Programmers write an application with two portions of code — functions to be executed on the CPU host and functions to be executed on the GPU device. The entry functions of the device code are tagged with a `_global_` keyword, and are referred to as *kernels*. A kernel executes in parallel across a set of parallel threads in a Single Instruction Multiple Thread (SIMT) model [1]. Since the host and device codes execute in two different memory spaces, the host code must include special calls for host-to-device and device-to-host data transfers. Figure 1 shows the sequence of steps involved in a typical CUDA kernel invocation.

3 The JCUDA Programming Interface and Compiler

With the availability of CUDA as an interface for C programmers, the natural extension for Java programmers is to use the Java Native Interface (JNI) as a bridge to CUDA via C. However, as discussed in Section 3.1, this approach is

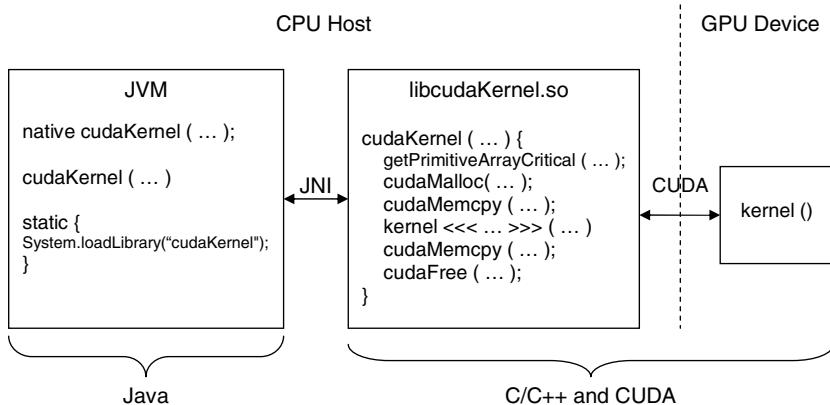


Fig. 2. Development process for accessing CUDA via JNI

neither easy nor productive. Sections 3.2 and 3.3 describe our JCUDA programming interface and compiler, and Section 3.4 summarizes our handling of Java arrays as parameters in kernel calls.

3.1 Current Approach: Using CUDA via JNI

Figure 2 summarizes the three-stage process that a Java programmer needs to follow to access CUDA via JNI today. It involves writing Java code and JNI stub code in C for execution on the CPU host, as well as CUDA code for execution on the GPU device. The stub code also needs to handle allocation and freeing of data in device memory, and data transfers between the host and device. It is clear that this process is tedious and error-prone, and that it would be more productive to use a compiler or programming tool that automatically generates stub code and data transfer calls.

3.2 The JCUDA Programming Interface

The JCUDA model is designed to be a programmer-friendly foreign function interface for invoking CUDA kernels from Java code, especially for programmers who may be familiar with Java and CUDA but not with JNI. We use the example in Figure 3 to illustrate JCUDA syntax and usage. The interface to external CUDA functions is declared in lines 90–93, which contain a static library definition using the `lib` keyword. The two arguments in a `lib` declaration specify the name and location of the external library using string constants. The library definition contains declarations for two external functions, `foo1` and `foo2`. The `acc` modifier indicates that the external function is a CUDA-accelerated kernel function. Each function argument can be declared as `IN`, `OUT`, or `INOUT` to indicate if a data transfer should be performed before the kernel call, after the kernel call or both. These modifiers allows the responsibility of device memory allocation and data transfer to be delegated to the JCUDA compiler. Our current

```

1  double[ ][ ] L_a = new double[NUM1][NUM2];
2  double[ ][ ][ ] L_aout = new double[NUM1][NUM2][NUM3];
3  double[ ][ ] Laex = new double[NUM1][NUM2];
4
5  initArray(L_a); initArray(L_aex); //initialize value in array
6
7  int [ ] ThreadsPerBlock = {16, 16, 1};
8  int [ ] BlocksPerGrid = new int[3]; BlocksPerGrid[3] = 1;
9  BlocksPerGrid[0] = (NUM1 + ThreadsPerBlock[0] - 1) / ThreadsPerBlock[0];
10 BlocksPerGrid[1] = (NUM2 + ThreadsPerBlock[1] - 1) / ThreadsPerBlock[1];
11
12 /* invoke device on this block/thread grid */
13 cudafoo.foo1 <<<< BlocksPerGrid, ThreadsPerBlock >>> (L_a, L_aout, L_aex);
14 printArray(L_a); printArray(L_aout); printArray(L_aex);
15
16 ...
17 ...
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
}

```

Fig. 3. JCUDA example

JCUDA implementation only supports scalar primitives and rectangular arrays of primitives as arguments to CUDA kernels. The OUT and INOUT modifiers are only permitted on arrays of primitives, not on scalar primitives. If no modifier is specified for an argument, it is default to be IN. As discussed later in Section 5, there are related approaches to CUDA language bindings that support modifiers such as IN and OUT, and the upcoming PGI 8.0 C/C++ compiler also uses an acc modifier to declare regions of code in C programs to be accelerated. To the best of our knowledge, none of the past efforts support direct invocation of user-written CUDA code from Java programs, with automatic support for data transfer (including copying of multidimensional Java arrays).

Line 13 shows a sample invocation of the CUDA kernel function `foo1`. Similar to CUDA's C interface, we use the <<<<...>>>¹ to identify a kernel call. The geometries for the CUDA grid and blocks are specified using two three-element integer arrays, `BlocksPerGrid` and `ThreadsPerBlock`. In this example, the kernel will be executed with $16 \times 16 = 256$ threads per block and by a number of blocks per grid that depends on the input data size (`NUM1` and `NUM2`).

3.3 The JCUDA Compiler

The JCUDA compiler performs source-to-source translation of JCUDA programs to Java program. Our implementation is based on Polyglot [10], a compiler front end for the Java programming language. Figures 4 and 5 show the Java static class declaration and the C glue code generated from the lib declaration in Figure 3. The Java static class introduces declarations with mangled names for native functions corresponding to JCUDA functions `foo1` and `foo2` respectively, as well as a static class initializer to load the stub library. In addition, three

¹ We use four angle brackets instead of three as in CUDA syntax because the “>>>” is already used as unsigned right shift operator in Java programming language.

```

private static class cudafoo {

    native static void HelloL_00024cudafoo_foo1(double[ ][ ] a,
        int[ ][ ] aout, float[ ][ ] aex, int[ ] dimGrid, int[ ] dimBlock, int sizeShared);

    static void foo1(double[ ][ ] a, int[ ][ ][ ] aout, float[ ][ ][ ] aex, int[ ] dimGrid, int[ ] dimBlock, int sizeShared) {
        HelloL_00024cudafoo_foo1(a, aout, aex, dimGrid, dimBlock, sizeShared);
    }

    native static void HelloL_00024cudafoo_foo2(short[ ][ ] a,
        double[ ][ ][ ] aex, int total, int[ ] dimGrid, int[ ] dimBlock, int sizeShared);

    static void foo2(short[ ][ ] a, double[ ][ ][ ] aex, int total, int[ ] dimGrid, int[ ] dimBlock, int sizeShared) {
        HelloL_00024cudafoo_foo2(a, aex, total, dimGrid, dimBlock, sizeShared);
    }

    static { java.lang.System.loadLibrary("HelloL_00024cudafoo_stub"); }
}

```

Fig. 4. Java static class declaration generated from `lib` definition in Figure 3

```

extern __global__ void foo1(double * d_a, signed int * d_aout, float * d_aex);

JNIEXPORT void JNICALL
Java_HelloL_00024cudafoo_HelloL_100024cudafoo_1foo1(JNIEnv *env, jclass cls, jobjectArray a,
    jobjectArray aout, jobjectArray aex, jintArray dimGrid, jintArray dimBlock, int sizeShared) {
    /* copy array a to the device */
    int dim_a[3] = {2};
    double * d_a = (double*) copyArrayJVMToDevice(env, a, dim_a, sizeof(double));

    /* Allocate array aout on the device */
    int dim_aout[4] = {3};
    signed int * d_aout = (signed int*) allocArrayOnDevice(env, aout, dim_aout, sizeof(signed int));

    /* copy array aex to the device */
    int dim_aex[3] = {2};
    float * d_aex = (float*) copyArrayJVMToDevice(env, aex, dim_aex, sizeof(float));

    /* Initialize the dimension of grid and block in CUDA call */
    dim3 d_dimGrid; getCUDA3Dim3(env, dimGrid, &d_dimGrid);
    dim3 d_dimBlock; getCUDA3Dim3(env, dimBlock, &d_dimBlock);

    foo1 <<< d_dimGrid, d_dimBlock, sizeShared >>> ((double *)d_a, (signed int *)d_aout, (float *)d_aex);

    /* Free device memory d_a */
    freeDeviceMem(d_a);

    /* copy array d_aout->aout from device to JVM, and free device memory d_aout */
    copyArrayDeviceToJVM(env, d_aout, aout, dim_aout, sizeof(signed int));
    freeDeviceMem(d_aout);

    /* copy array d_aex->aex from device to JVM, and free device memory d_aex */
    copyArrayDeviceToJVM(env, d_aex, aex, dim_aex, sizeof(float));
    freeDeviceMem(d_aex);

    return;
}

```

Fig. 5. C glue code generated for the `foo1` function defined in Figure 3

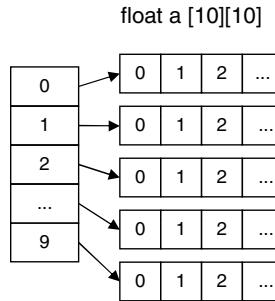


Fig. 6. Java Multidimensional Array Layout

parameters are added to each call — *dimGrid*, *dimBlock*, and *sizeShared* — corresponding to the grid geometry, block geometry, and shared memory size. As we can see in Figure 5, the generated C code inserts host-device data transfer calls in accordance with the IN, OUT and INOUT modifiers in Figure 3.

3.4 Multidimensional Array Issues

A multidimensional array in Java is represented as an array of arrays. For example, a two-dimensional float array is represented as a one-dimensional array of objects, each of which references a one-dimensional float array as shown in the Figure 6. This representation supports general nested and ragged arrays, as well as the ability to pass subarrays as parameters while still preserving pointer safety. However, it has been observed that this generality comes with a large overhead for the common case of multidimensional rectangular arrays [11].

In our work, we focus on the special case of dense rectangular multidimensional arrays of primitive types as in C and Fortran. These arrays are allocated as nested arrays in Java and as contiguous arrays in CUDA. The JCUDA runtime performs the necessary gather and scatter operations when copying array data between the JVM and the GPU device. For example, to copy a Java array of `double[20][40][80]` from the JVM to the GPU device, the JCUDA runtime makes $20 \times 40 = 800$ calls to the CUDA `cudaMemcpy` memory copy function with 80 double-words transferred in each call. In future work, we plan to avoid this overhead by using X10's multidimensional arrays [12] with contiguous storage of all array elements, instead of Java's multidimensional arrays.

4 Performance Evaluation

4.1 Experimental Setup

We use four Section 2 benchmarks from the Java Grande Forum (JGF) Benchmarks [34] to evaluate our JCUDA programming interface and compiler — Fourier coefficient analysis (Series), Sparse matrix multiplication (Sparse), Successive over-relaxation (SOR), and IDEA encryption (Crypt). Each of these benchmarks has

three problem sizes for evaluation — A, B and C — with Size A being the smallest and Size C the largest. For each of these benchmarks, the compute-intensive portions were rewritten in CUDA whereas the rest of the code was retained in its original Java form except for the JCUDA extensions used for kernel invocation. The rewritten CUDA codes are parallelized in the same way as the original Java multithreaded code, with each CUDA thread performing the same computation as a Java thread.

The GPU used in our performance evaluations is a NVIDIA Tesla C1060 card, containing a GPU with 240 cores in 30 streaming multiprocessors, a 1.3 GHz clock speed, and 4GB memory. It also supports double-precision floating-point operations, which was not available in earlier GPU products from NVIDIA. All benchmarks were evaluated with double-precision arithmetic, as in the original Java versions. The CPU hosting this GPGPU is an Intel Quad-Core CPU with a 2.83GHz clock speed, 12MB L2 Cache and 8GB memory. The software installations used include a Sun Java HotSpot 64-bit virtual machine included in version 1.6.0_07 of the Java SE Development Kit (JDK), version 4.2.4 of the GNU Compiler Collection (gcc), version 180.29 of the NVIDIA CUDA driver, and version 2.0 of the NVIDIA CUDA Toolkit.

There are two key limitations in our JCUDA implementation which will be addressed in future work. First, as mentioned earlier, we only support primitives and rectangular arrays of primitives as function arguments in the JCUDA interface. Second, the current interface does not provide any direct support for reuse of data across kernel calls since the parameter modes are restricted to IN, OUT and INOUT.

4.2 Evaluation and Analysis

Table I shows the execution times (in seconds) of the Java and JCUDA versions for all three data sizes of each of the four benchmarks. The Java execution times

Table 1. Execution times in seconds, and Speedup of JCUDA relative to 1-thread Java executions (30 blocks per grid, 256 threads per block)

Benchmark	Series			Sparse		
	A	B	C	A	B	C
Data Size						
Java-1-thread execution time	7.6s	77.42s	1219.40s	0.50s	1.17s	19.87s
Java-2-threads execution time	3.84s	39.21s	755.05s	0.26s	0.54s	8.68s
Java-4-threads execution time	2.03s	19.82s	390.98s	0.25s	0.39s	5.32s
JCUDA execution time	0.11s	1.04s	10.14s	0.07s	0.14s	0.93s
JCUDA Speedup w.r.t. Java-1-thread	67.26	74.51	120.32	7.25	8.30	21.27

Benchmark	SOR			Crypt		
	A	B	C	A	B	C
Data Size						
Java-1-thread execution time	0.62s	1.60s	2.82s	0.51s	3.26s	8.16s
Java-2-threads execution time	0.26s	1.32s	2.59s	0.27s	1.65s	4.10s
Java-4-threads execution time	0.16s	1.37s	2.70s	0.11s	0.21s	2.16s
JCUDA execution time	0.09s	0.21s	0.37s	0.02s	0.16s	0.45s
JCUDA Speedup w.r.t. Java-1-thread	6.74	7.73	7.70	22.17	20.12	17.97

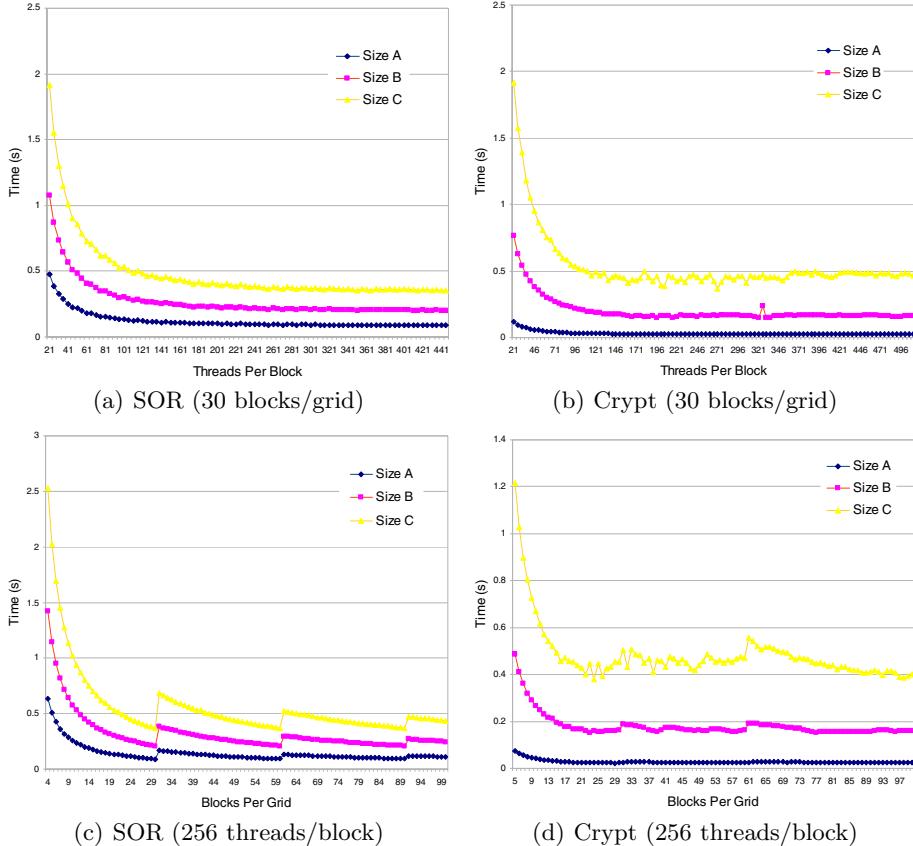


Fig. 7. Execution time by varying threads/block and blocks/grid in CUDA kernel invocations on the Tesla C1060 GPU

were obtained for 1, 2 and 4 threads, representing the performance obtained by using 1, 2, and 4 CPU cores respectively. The JCUDA execution times were obtained by using a single Java thread on the CPU combined with multiple threads on the GPU in a configuration of 256 threads per block and 30 blocks per grid. The JCUDA kernel was repeatedly executed 10 times, and the fastest times attained for each benchmark and test size are listed in the table.

The bottom row shows the speedup of the JCUDA version relative to the 1-thread Java version. The results for Size C (the largest data size) show speedups ranging from $7.70\times$ to $120.32\times$, whereas smaller speedups were obtained for smaller data sizes — up to $74.51\times$ for Size B and $67.26\times$ for Size A. The benchmark that showed the smallest speedup was SOR. This is partially due to the fact that our CUDA implementation of the SOR kernel performs a large number of barrier (`__syncthreads`) operations — 400, 600, and 800 for sizes A, B, and C respectively. In contrast, the Series examples show the largest speedup because it represents an embarrassingly parallel application with no communication or

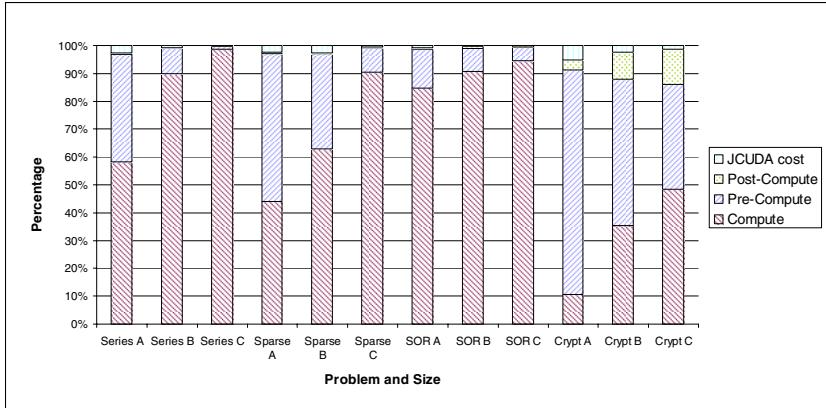


Fig. 8. Breakdown of JCUDA Kernel Execution Times

synchronization during kernel execution. In general, we see that the JCUDA interface can deliver significant speedups for Java applications with double-precision floating point arithmetic by using GPGPUs to offload computations.

Figure 7 shows the results of varying threads/block and blocks/grid for SOR and Crypt to study the performance variation relative to the geometry assumed in Table I (256 threads/block and 30 blocks/grid). In Figures 7(a) and 7(b), for 30 blocks per grid, we see that little performance gain is obtained by increasing the number of threads per block beyond 256. This suggests that 256 threads per block (half the maximum value of 512) is a large enough number to enable the GPU to maintain high occupancy of its cores [13]. Next, in Figures 7(c) and 7(d), for 256 threads per block, we see that execution times are lowest when the blocks/grid is a multiple of 30. This is consistent with the fact that the Tesla C1060 has 30 streaming multiprocessors (SM). Further, we observe a degradation when the blocks/grid is 1 more than a multiple of 30 (31, 61, 91, ...) because of the poor load balance that results on a 30-SM GPU processor.

Figure 8 shows the percentage breakdown of the JCUDA kernel execution times into *compute*, *pre-compute*, *post-compute* and *JCUDA cost* components. The *compute* component is the time spent in kernel execution on the GPGPU. The *pre-compute* and *post-compute* components represent the time spent in data transfer before and after kernel execution. The *JCUDA cost* includes the overhead of the JNI calls generated by the JCUDA kernel. As the figure shows, for each benchmark, as the problem becomes larger, the *compute* time percentage increases. The *JCUDA* overhead has minimal impact on performance, especially for Size C executions. The impact of *pre-compute* and *post-compute* times depends on the communication requirements of individual benchmarks.

5 Related Work

In this section, we briefly discuss two areas of related work in making CUDA kernels accessible to other languages. The first approach is to provide library

bindings for the CUDA API. Examples of this approach include the JCublas and JCufft [14] Java libraries that provide Java bindings to the standard CUBLAS and CUFFT libraries from CUDA. We recently learned that JCublas and JCufft are part of a new Java library called jCUDA. In contrast, the JCUDA interface introduced in this paper is a language-based approach, in which the compiler automatically generates glue code and data transfers.

PyCuda [15] from Python community is a Python binding for the CUDA API. It also allows CUDA kernel code to be embedded as a string in a Python program. RapidMind provides a multicore programming framework based on the C++ programming language [16]. A programmer can embed a kernel intended to run on the GPU as a delimited piece of code directly in the program. RapidMind also supports IN, and OUT keywords for function parameters to guide the compiler in generating appropriate code for data movement.

The second approach is to use compiler parallelization to generate CUDA kernel code. A notable example of this approach can be found in the PGI 8.0 x64+GPU compilers for C and Fortran which allow programmers to use directives, such as *acc*, *copyin*, and *copyout*, to specify regions of code or functions to be GPU-accelerated. The compiler splits portions of the application between CPU and GPU based on these user-specified directives and generates object codes for both CPU and GPUs from a single source file. Another related approach is to translate OpenMP parallel code to hybrid code for CPUs and GPUs using compiler analysis, transformation and optimization techniques [17]. A recent compiler research effort that is of relevance to Java can be found in [18], where the Jikes RVM dynamic optimizing compiler [19] is extended to perform automatic parallelization at the bytecode level.

To the best of our knowledge, none of the past efforts support direct invocation of user-written CUDA code from Java programs, with automatic support for data transfer of primitives and multidimensional arrays of primitives.

6 Conclusions and Future Work

In this paper, we presented the JCUDA programming interface that can be used by Java programmers to invoke CUDA kernels without having to worry about the details of JNI calls and data transfers (especially for multidimensional arrays) between the host and device. The JCUDA compiler generates all the necessary JNI glue code, and the JCUDA runtime handles data transfer before and after each kernel call. Our preliminary performance results obtained on four Java Grande benchmarks show significant performance improvements of the JCUDA-accelerated versions over their original versions. The results for Size C (the largest data size) showed speedups ranging from $6.74\times$ to $120.32\times$ with the use of a GPGPU, relative to CPU execution on a single thread. We also discussed the impact of problem size, communication overhead, and synchronization overhead on the speedups that were obtained.

To address the limitations listed in Section 4.1, we are considering to add a KEEP modifier for an argument to specify GPU resident data between kernel

calls. To support the overlapping of computation and data transfer, we are developing memory copy primitives for programmers to initiate asynchronous data copy between CPU and GPU. Those primitives can use either the conventional *cudaMemcpy* operation or the page-locked memory mapping mechanism introduced in the latest CUDA development kit. In addition to those, there are also several interesting directions for our future research. We plan to use the JCUDA interface as a target language to support high-level parallel programming language like X10 [12] and Habanero-Java in the Habanero Multicore Software Research project [20]. In this approach, we envision generating JCUDA code and CUDA kernels from a single source multi-place X10 program. Pursuing this approach will require solving some interesting new research problems such as ensuring that offloading X10 code onto a GPGPU will not change the exception semantics of the program. Another direction that we would like to explore is the use of the language extensions recommended in [21] to simplify automatic parallelization and generation of CUDA code.

Acknowledgments

We are grateful to Jisheng Zhao and Jun Shirako for their help with the Polyglot compiler infrastructure and JGF benchmarks. We would also like to thank Tim Warburton and Jeffrey Bridge for their advice on constructing the GPGPU system used to obtain the experimental results reported in this paper.

References

1. Nickolls, J., Buck, I., Garland, M., Nvidia, Skadron, K.: Scalable Parallel Programming with CUDA. *ACM Queue* 6(2), 40–53 (2008)
2. Java Grande Forum Panel, Java Grande Forum Report: Making Java Work for High-End Computing, Java Grande Forum, SC 1998, Tech. Rep. (November 1998)
3. Bull, J.M., Smith, L.A., Pottage, L., Freeman, R.: Benchmarking Java Against C and Fortran for Scientific Applications. In: *Proceedings of the 2001 joint ACM-ISCOPE Conference on Java Grande*, pp. 97–105. ACM Press, New York (2001)
4. Smith, L.A., Bull, J.M., Obdrzálek, J.: A Parallel Java Grande Benchmark Suite. In: *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, p. 8. ACM Press, New York (2001)
5. SciMark Java Benchmark for Scientific and Numerical Computing, <http://math.nist.gov/scimark2/>
6. Liang, S.: Java Native Interface: Programmer’s Guide and Specification. Sun Microsystems (1999)
7. Carpenter, B., Getov, V., Judd, G., Skjellum, A., Fox, G.: MPJ: MPI-Like Message Passing for Java. *Concurrency - Practice and Experience* 12(11), 1019–1038 (2000)
8. AMD, ATI Stream Computing - Technical Overview. AMD, Tech. Rep. (2008)
9. Khronos OpenCL Working Group, The OpenCL Specification - Version 1.0. The Khronos Group, Tech. Rep. (2009)
10. Nystrom, N., Clarkson, M.R., Myers, A.C.: Polyglot: An Extensible Compiler Framework for Java. In: Kahng, H.-K. (ed.) *ICOIN 2003. LNCS*, vol. 2662, pp. 138–152. Springer, Heidelberg (2003)

11. Moreira, J.E., Midkiff, S.P., Gupta, M., Artigas, P.V., Snir, M., Lawrence, R.D.: Java Programming for High-Performance Numerical Computing. *IBM Systems Journal* 39(1), 21–56 (2000)
12. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioğlu, K., von Praun, C., Sarkar, V.: X10: an Object-Oriented Approach to Non-Uniform Cluster Computing. In: *OOPSLA 2005: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pp. 519–538. ACM, New York (2005)
13. NVIDIA, NVIDIA CUDA Programming Guide 2.2.plus 0.5em minus 0.4emNVIDIA (2009), <http://www.nvidia.com/cuda>
14. Java Binding for NVIDIA CUDA BLAS and FFT Implementation, <http://www.jcuda.de>
15. PyCuda, <http://documen.tician.de/pycuda/>
16. Matthew Monteyne, RapidMind Multi-Core Development Platform. RapidMind Inc., Tech. Rep (2008)
17. Lee, S., Min, S.-J., Eigenmann, R.: Openmp to gpgpu: a compiler framework for automatic translation and optimization. In: *PPoPP 2009: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 101–110. ACM, New York (2009)
18. Leung, A.C.-W.: Thesis: Automatic Parallelization for Graphics Processing Units in JikesRVM. University of Waterloo, Tech. Rep. (2008)
19. Alpern, B., Augart, S., Blackburn, S.M., Butrico, M., Cocchi, A., Cheng, P., Dolby, J., Fink, S., Grove, D., Hind, M., McKinley, K.S., Mergen, M., Moss, J.E.B., Ngo, T., Sarkar, V.: The Jikes Research Virtual Machine Project: Building an Open-Source Research Community. *IBM Systems Journal* 44(2), 399–417 (2005)
20. Habanero Multicore Software Project, <http://habanero.rice.edu>
21. Shirako, J., Kasahara, H., Sarkar, V.: Language Extensions in Support of Compiler Parallelization. In: Adve, V., Garzarán, M.J., Petersen, P. (eds.) *LCPC 2007. LNCS*, vol. 5234, pp. 78–94. Springer, Heidelberg (2008)

Fast and Efficient Synchronization and Communication Collective Primitives for Dual Cell-Based Blades*

Epifanio Gaona, Juan Fernández, and Manuel E. Acacio

Dept. de Ingeniería y Tecnología de Computadores, Universidad de Murcia, Spain
{fanios.gr, juanf, meacacio}@ditec.um.es

Abstract. The Cell Broadband Engine (Cell BE) is a heterogeneous multi-core processor specifically designed to exploit thread-level parallelism. Its memory model comprehends a common shared main memory and eight small private local memories. Programming of the Cell BE involves dealing with multiple threads and explicit data movement strategies through DMAs which make the task very challenging. This situation gets even worse when dual Cell-based blades are considered. In this context, fast and efficient collective primitives are indispensable to reduce complexity and optimize performance.

In this paper, we describe the design and implementation of three collective operations: barrier, broadcast and reduce. Their design takes into consideration the architectural peculiarities and asymmetries of dual Cell-based blades. Meanwhile, their implementation requires minimal resources, a signal register and a buffer. Experimental results show low latencies and high bandwidths, synchronization latency of 637 ns, broadcast bandwidth of 38.33 GB/s for 16 KB messages, and reduce latency of 1535 ns with 32 *floats*, on a dual Cell-based blade with 16 SPEs.

1 Introduction

The Cell BE was jointly developed by Sony, Toshiba and IBM to provide improved performance on game and multimedia applications [1]. However, there is a growing interest in using the Cell BE for high-performance computing due to its tremendous potential in terms of theoretical peak performance. From the architectural point of view, the Cell BE can be classified as a heterogeneous multi-core processor specifically designed to exploit thread-level parallelism. As for its memory model, the Cell BE comes with a hybrid scheme with a common shared main memory and eight small private local memories. The combination of all these factors makes programming of the Cell BE a really complex task.

* This work has been jointly supported by the Spanish MEC under grants “TIN2006-15516-C04-03” and European Comission FEDER funds under grant “Consolider Ingenio-2010 CSD2006-00046”. Epifanio Gaona is supported by fellowship 09503/FPI/08 from Comunidad Autónoma de la Región de Murcia (Fundación Séneca, Agencia Regional de Ciencia y Tecnología).

Cell BE programmers must explicitly cope with multiple threads that have to orchestrate frequent data movements to overlap computation and communication due to the small size of the private local memories [2].

In this scenario, a number of programming models and platforms have been proposed for the Cell BE. Some of them are based on well-known shared-memory and message-passing libraries such as OpenMP [3] and MPI [4,5]. In the meantime, others such as CellSs [6] and RapidMind [7] are based on task-parallel and stream programming models, respectively. Anyway, synchronization and coordination of multiple threads is a common source of programming errors and performance bottlenecks [8]. For that reason, these programming models either offer explicit synchronization and communication collective primitives, such as `MPI_Barrier` or `#pragma omp barrier`, in order to make code less error prone. In any case, fast and efficient collective primitives are clearly needed.

The Cell BE provides programmers with a broad variety of communication and synchronization primitives between the threads that comprise parallel applications, such as DMAs, mailboxes, signals and atomic operations, which were evaluated by us in [9] for dual Cell-based blades. Nevertheless, the Cell BE SDK provided by IBM does not provide programmers with synchronization and communication collective primitives similar to those available for many other parallel systems [10]. To the best of our knowledge, the work presented in [11] is the only one that tackles this problem. However, it focuses on the MPI programming model for the Cell BE introduced by [5] considering that application data is stored in main memory and is not optimized for dual Cell-based Blades. In contrast, our proposal in this work is more general and efficient. First, it assumes that data resides in the SPEs' LSs. Second, it has been specifically optimized for dual Cell-based blades taking advantage of the know-how acquired in [9] and, therefore, improves barrier performance results (see Section 4). Third, it requires less resources to implement the very same set of collective primitives.

Our main contributions are: (1) a description of the design of several algorithms for three common collective operations: barrier, broadcast and reduce;

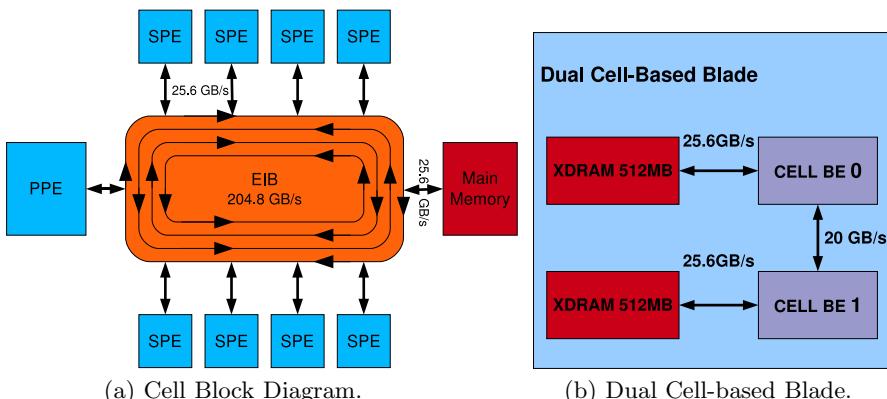


Fig. 1. Cell BE Architecture

(2) an optimized implementation of the algorithms for dual Cell-based blades requiring minimal resources, a signal register and a single 16 KB buffer; and, (3) a comparative analysis of the performance of the algorithms. Note that the use of a single 16 KB buffer limits the maximum broadcast and reduce sizes. This limitation is not such a restriction due to the small size of the LSs and the fact that reductions involving simple operations on small data sizes are the prevalent case in most scientific applications [12]. Finally, as an outcome of this work, Cell BE programmers should be able to implement the most appropriate version of these primitives depending of their expertise and performance requirements.

The rest of the paper is organized as follows. In Section 2 we provide a revision of the architecture and programming of dual Cell-based blades. Next, in Section 3 our proposal for designing and implementing the three collective primitives is explained. Then, the experimental results obtained on a dual Cell-based blade are presented in Section 4. Finally, Section 5 gives the main conclusions of this work.

2 Cell BE: Architecture and Programming

The Cell Broadband Engine (Cell BE) [1] is an heterogeneous multi-core chip composed of one general-purpose processor, called *PowerPC Processor Element* (PPE), eight specialized co-processors, called *Synergistic Processing Elements* (SPEs), a high-speed memory interface controller, and an I/O interface, all integrated in a single chip. All these elements communicate through an internal high-speed *Element Interconnect Bus* (EIB) (see Figure 1(a)).

A dual Cell-based blade is composed of two separate Cell BEs linked together through their EIBs as shown in Figure 1(b). The EIB is extended transparently across a high-speed coherent interface running at 20 GBytes/second in each direction [2]. In this configuration the two Cell BEs operate in SMP mode with full cache and memory coherency.

The Cell BE provides programmers with a variety of synchronization and communication mechanisms: *DMA Transfers*, *Atomic Operations*, *Mailboxes* and *Signals*.

SPEs use DMA transfers to read from (GET) or write to (PUT) main memory, or to copy data between the eight SPEs' LSs (MOV). DMA transfer size must be 1, 2, 4, 8 or a multiple of 16 Bytes up to a maximum of 16 KB. DMA transfers can be either blocking or non-blocking. The latter allow to overlap computation and communication: there might be up to 128 simultaneous transfers between the eight SPEs' LSs and main memory. In all cases, peak performance can be achieved when both the source and destination addresses are 128-Byte aligned and the size of the transfer is an even multiple of 128 Bytes [13].

Read-modify-write atomic operations enable simple transactions on single words residing in main memory. For example, the *atomic_add_return* atomic operation adds a 32-bit integer to a word in main memory and returns its value before the addition.

Mailboxes are FIFO queues that support exchange of 32-bit messages among the SPEs and the PPE. Each SPE includes two outbound mailboxes, called *SPU Write Outbound Mailbox* and *SPU Write Outbound Interrupt Mailbox*, to send messages from the SPE; and a 4-entry inbound mailbox, called *SPU Read Inbound Mailbox*, to receive messages.

In contrast, signals were designed with the only purpose of sending notifications to the SPEs. Each SPE has two 32-bit signal registers to collect incoming notifications, namely *SPU Signal Notification 1 (SIG1)* and *SPU Signal Notification 2 (SIG2)*. A signal register is assigned a MMIO register to enable remote SPEs and the PPE to send individual signals (*overwrite mode*) or combined signals (*OR mode*) to the owner SPE. It is worth noting that the latter mode allows to collect and identify incoming notifications from remote SPEs. To do so, it is enough to assign the i th bit of a signal register to the i th SPE, that is, SPE i can signal SPE j by sending the value 2^{id} which would set the i th bit of the target SPE j 's signal register. In this way, SPE j can determine the number and identity of the remote SPEs by simply using bit masks. This strategy revealed extremely useful to implement collective primitives as we will see in Section 3.

Cell BE programming requires separate programs, written in C/C++, for the PPE and the SPEs, respectively. The PPE program can include extensions (e.g., `vec_add`), to use its VMX unit; and library function calls, to manage threads and perform communication and synchronization operations (e.g., `spe_create_thread` and `spe_write_in mbox`). The SPE program follows an SPMD model. It includes extensions, to execute SIMD instructions, and communication and synchronization operations (e.g., `spu_add` and `mfc_get`).

Programming of a dual Cell-based blade is similar to that of an independent Cell from a functional point of view. However, there are two important differences. Firstly, dual Cell-based blades have 16 SPEs at programmer's disposal rather than 8 SPEs. This feature doubles the maximum theoretical peak performance but also makes much more difficult to fully exploit thread-level parallelism. Secondly, from an architectural point of view, synchronization and communication operations crossing the inter-Cell interface result in significantly less performance than those that stay on-chip [9]. This feature is a key factor that must be taken into consideration to avoid unexpected and undesirable surprises when designing synchronization and communication collective primitives for a dual Cell-based blade platform.

3 Design and Implementation

This section describes the design and implementation of several alternative algorithms for three common collective synchronization and communication primitives: barrier, broadcast and reduce. Most of them are based on a three-phase scheme. In the *ready* phase all group members wait for each other to reach the collective primitive call. The *transfer* phase is executed in between the other two phases and is devoted to transferring data among SPEs' LSs using DMAs when necessary. Finally, in the *go* phase all group members are informed to resume

computation. Unless otherwise noted, only one signal register in *OR mode* is needed to implement the *ready* and *go* phases as described in Section 2. Moreover, even though it is not discussed in the paper, the 16 most significant bits (*msb*) and the 16 less significant bits (*lsb*) of such a signal register have been used alternatively in order to ensure correctness between consecutive executions of each primitive when necessary. Note that the second signal register could be used for synchronizing up to 4 Cell chips. A single 16 KB buffer is in turn employed in the *transfer* phase. Finally, in all cases, there is a single group that comprehends all threads run by the SPEs used by the application.

3.1 Barrier

A barrier synchronization blocks the callers until all group members have invoked the primitive. We have implemented three different algorithms and two variants for the first two ones (see Figure 2).

All-To-One Barrier (ATOBar). In the *ready* phase every SPE signals the root SPE (see the arrows labeled with number 1 in Figure 2(a)). The root SPE in turn waits for all SPEs to enter the barrier and then clears its signal register. In the *go* phase, the root SPE signals all SPEs (see the arrows marked with number 2 in Figure 2(b)). As SPEs receive the acknowledgement from the root SPE, they resume computation.

All-To-One Barrier for Blades (ATOBarB). This algorithm introduces an optimization over the previous one in order to reduce the inter-Cell traffic when the second Cell comes into play. To do so, the ATOBar algorithm is executed locally by both Cells so that both root SPEs must signal each other (inter-Cell synchronization) before the beginning of the *go* phase. Consequently, only two signals are sent through the inter-Cell interface (see the arrows marked with number 2 in Figure 2(b)).

Tree Topology Barrier (TTBar). This scheme is similar to the ATOBar algorithm but a tree structure is used in the upward and downward directions for the *ready* and *go* phases, respectively (see Figure 2(c)). The algorithm requires $2 \times \lceil \log_d N \rceil$ steps where d is the degree of the tree.

Tree Topology Barrier for Blades (TTBarB). This version follows the same idea as the ATOBarB algorithm but using instead the TTBar algorithm for synchronization inside each Cell (as illustrated in Figure 2(d)). In this case, the algorithm employs $2 \times \lceil \log_d \frac{N}{2} \rceil + 1$ steps.

Pairwise Exchange Barrier (PEBar). Parwise exchange is a well-known recursive algorithm [14]. Unlike the previous schemes, all SPEs are progressively paired up to synchronize in each round. Let N be the number of SPEs. At step s , SPE i and SPE j , where $j = i \oplus 2^s$, signal each other. If the number of SPEs is a power of two, then the algorithm requires $\log_2 N$ steps. Otherwise, PE needs $\lceil \log_2 N \rceil + 2$ steps [14]. Let M be the largest power of two less than N . First, $\forall k \geq M$, SPE k signals SPE i , where $i = k - M$. Second, $\forall i < M$,

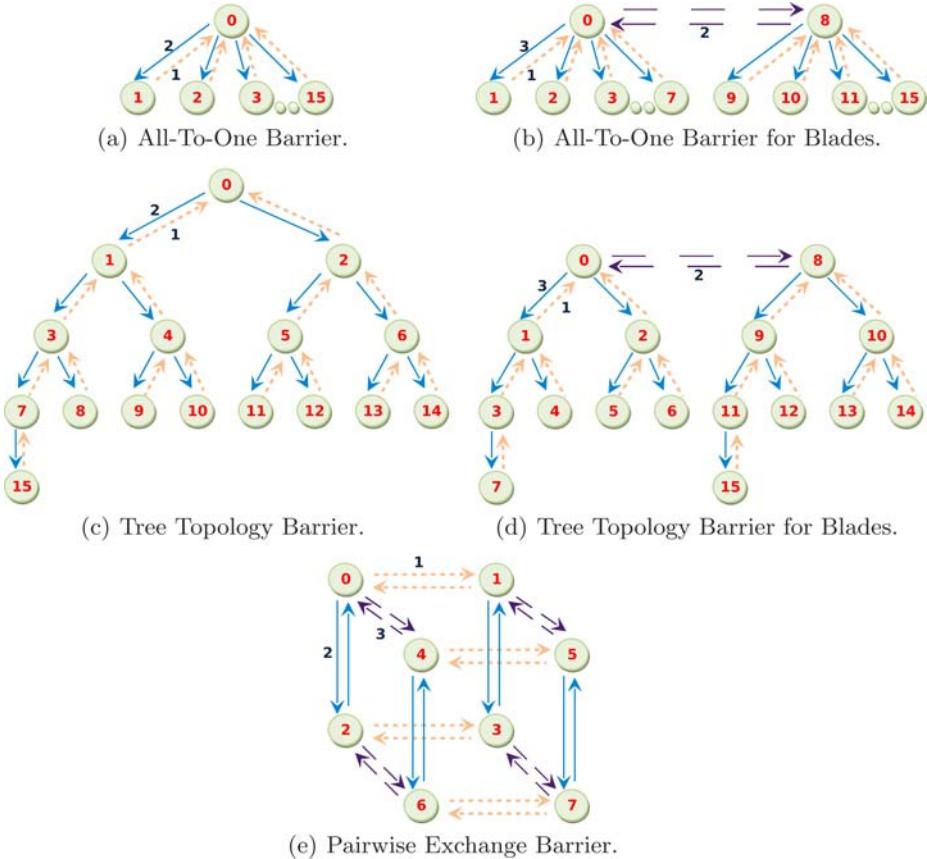


Fig. 2. Barrier Algorithms: *ready*, inter-Cell synchronization, and *go* phases

SPE i executes the PE algorithm. Third, $\forall k \geq M$, SPE i signals SPE k , where $i = k - M$, upon completion of the intermediate PE synchronization. Figure 2(e) shows these three steps.

3.2 Broadcast

A broadcast communication is similar to a barrier synchronization in which all group members must receive a message from the root member before they leave the barrier. The broadcast algorithms presented here are based on a three-phase scheme: *ready*, *broadcast* and *go*. The *ready* and *go* phases are similar to those of barrier algorithms. In the meantime, the *transfer* phase becomes a *broadcast* phase where a message is Moved from the source SPE's LS to the destination SPEs' LSs. The broadcast buffers always reside at the same LS locations in all SPEs. Thus, their effective addresses can be computed by just adding the effective starting address of the corresponding SPEs' LSs. This simplification

is not a hard constraint since different effective destination addresses could be easily gathered in the *ready* phase. Broadcast message size is limited to 16 KB, that is, the maximum DMA size, and a 16 KB buffer per SPE is hence needed. Note that broadcast messages greater than 16 KB could be sent by using DMA lists or by implementing a flow control mechanism (e.g. through mailboxes) in charge of monitoring the transmission of the segments of the message.

We have implemented two different algorithms (see Figure 3), the broadcast versions of the ATOBar and TTBar algorithms, with a number of variants each through incorporating some optimizations. PE broadcast version is not considered because the *ready* and *go* phases cannot be decoupled. Description of non-blade, unoptimized versions of ATOBcastB_{Opt} and TTBcastB_{Opt} have been omitted since their implementations are straightforward from the description of the corresponding barrier algorithms.

All-To-One Broadcast for Blades with Optimizations (ATOBcastB_{Opt}). This algorithm is similar to the ATOBar version with two major optimizations. On the one hand, both root SPEs run the ATOBcast algorithm but the root SPE on the second Cell is treated as another remote SPE of the root SPE on the first Cell. In this way, a single broadcast message and two signals go across the inter-Cell interface. On the other hand, the *broadcast* phase overlaps with the *ready* and *go* phases. When the root SPE is signaled by a remote SPE, it immediately starts an asynchronous DMA to transfer the broadcast message from the local LS to the corresponding remote SPE's LS. After signals from all remote SPEs have been received and the subsequent DMAs have been initiated, the root SPE waits for each DMA to complete and immediately signals the corresponding remote SPE, in the same order signals were received in the *ready* phase. As a minor optimization of the last step, the root SPE on the second Cell is given a highest priority than the other remote SPEs.

Tree Topology Broadcast for Blades with Optimizations (TTBcastB_{Opt}). This scheme is derived from the TTBar algorithm and incorporates similar optimizations to the ones used in ATOBcastB_{Opt} .

3.3 Reduce

A reduce communication behaves like a broadcast communication in which messages flow in the reverse order. The broadcast algorithms presented here are based on a three-phase scheme: *ready*, *gather* and *go*. The *ready* and *go* phases are identical to those of broadcast algorithms. In the *gather* phase incoming reduce messages, consisting of arrays of integers or single-precision floating-point numbers, are combined using a specific operator (e.g. ADD or SUB) through SIMD *intrinsics* (e.g. `spu_add` or `spu_sub`). Broadcast buffers are also used here for reduce operations. As in the case of broadcast, we have implemented two different algorithms (graphical representation would be equal to that of Figure 3 if arrows labeled with 2 appeared reversed), the reduce versions of the ATOBarB and TTBarB algorithms, with a number of variants each through incorporating the same optimizations used by the ATOBcastB_{Opt} and TTBcastB_{Opt} algorithms.

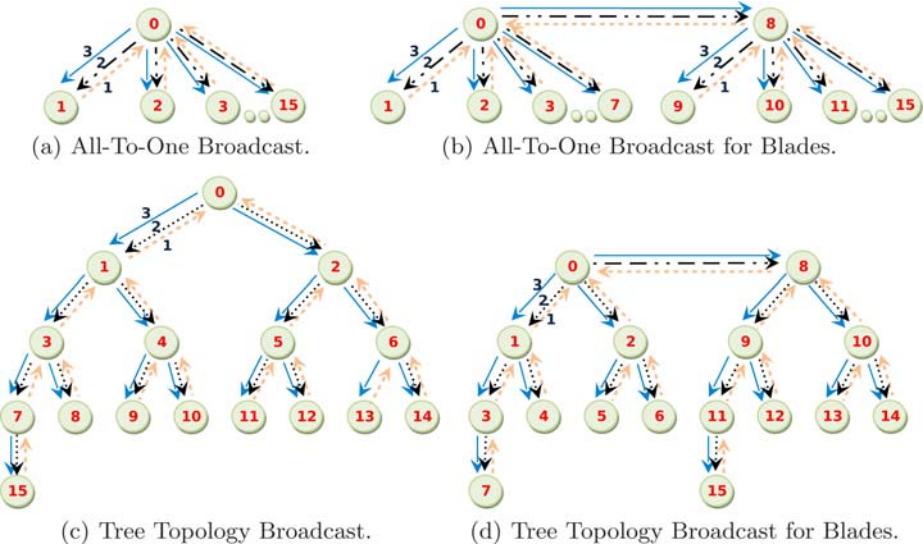
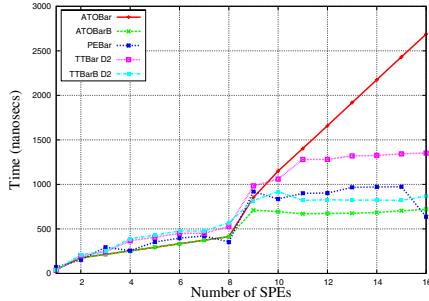
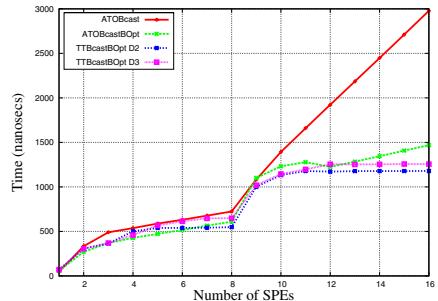


Fig. 3. Broadcast Algorithms: *ready*, *broadcast*, and *go* phases

Again, non-blade, unoptimized versions of the ATORedB_{Opt} and TTRedB_{Opt} algorithms have been left out since their implementations are straightforward from the description of the corresponding barrier algorithms.

All-To-One Reduce for Blades with Optimizations (ATORedB_{Opt}). This algorithm applies similar optimizations to the ones implemented by the ATOBcastB_{Opt} algorithm to ATORed. First, both root SPEs run the ATORed algorithm but the root SPE on the second Cell becomes another remote SPE of the root SPE on the first Cell. Thus, it is guaranteed that no more than a single reduce message and two signals cross the inter-Cell interface. Second, the *gather* phase overlaps with the *ready* and *go* phases. Upon signal reception, the root SPE initiates an asynchronous DMA to transfer the reduce message from the remote SPE's LS into its local LS. When all signals from children SPEs have been processed, the root SPE proceeds in the same order they were received: (1) waits for DMA completion, (2) computes the reduce using local data and remote data, and (3) signals the corresponding remote SPE. Also, the root SPE on the second Cell is given a highest priority as a minor optimization. Finally, reduce operations are limited to 512 integers or single-precision floating point numbers and, therefore, reduce messages from all remote SPEs can be accommodated in the same buffer used by broadcast operations.

Tree Topology Reduce for Blades with Optimizations (TTRedB_{Opt}). This scheme is derived from the TTRed algorithm and incorporates similar optimizations to the ones used in ATOBcastB_{Opt}. Consequently, regardless of the degree of the tree, the root SPE on the second Cell is treated as a child of the root SPE on the first Cell, and the *gather* phase overlaps with the *ready* and *go*

**Fig. 5.** Barrier Latency**Fig. 6.** Broadcast Latency (128 Bytes)

phases in the root and intermediate SPEs. In this case, reduce messages gathered from children SPEs are limited to 2,048 integers or single-precision floating point numbers in order not to exceed the maximum broadcast buffer size.

4 Performance Evaluation

This section details the making of the experiments and analyzes the experimental results obtained for the three collective primitives. In all experiments, the number of demanded SPEs invoke the specified collective primitive in a loop for one million iterations. Reported results are the best of three experiments executed in a row.

All the algorithms considered in this work have been implemented and evaluated using the IBM SDK v3.1 installed atop Fedora Core 9 on a dual Cell-based IBM BladeCenter QS20 blade which incorporates two 3.2 GHz Cell BEs v5.1, namely Cell0 and Cell1, with 1 GByte of main memory and a 670 GB hard disk.

4.1 Barrier

First of all, we present in Figure 5 the latency (measured in nanoseconds) for the implementations of the barrier primitive discussed in Subsection 3.1. As we can see, the naive All-To-One Barrier (ATOBar) is the best option when the number of SPEs involved is small (less than 8). In this case, all SPEs are in the same Cell chip and two signals, one from each SPE to the root SPE and another one in the opposite direction, suffice to complete the barrier primitive. Unfortunately, as the number of SPEs grows, so does the time taken by the ATOBar implementation, which becomes impractical when the number of SPEs is greater than 8. In this case, SPEs belonging to the two Cell chips participate in the barrier and latency is dominated by inter-chip signals (round-trip latencies of inter-chip and intra-chip signals are 619.4 ns and 160.1 ns [9], respectively). In order to cope with this problem, we proposed the All-To-One Barrier for Blades (ATOBarB) implementation. Up to 8 SPEs, the behavior of ATOBar and ATOBarB is exactly the same. From 9 SPEs up, two SPEs (one per Cell chip) act as roots and just two signals (exchanged by the two root SPEs once the local *ready* phases have been completed) must traverse the inter-Cell interface.

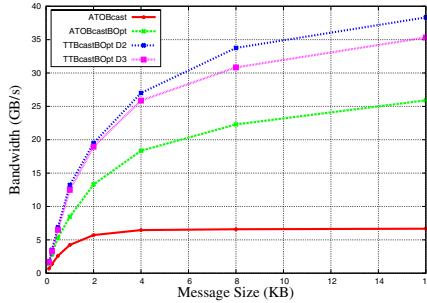


Fig. 7. Broadcast Bandwidth

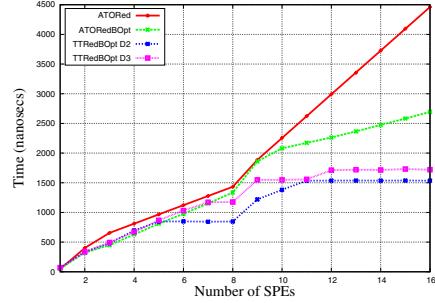


Fig. 8. Reduce Latency (32 floats)

As seen in Figure 5, this optimization ensures perfect scalability for more than 9 SPEs, since the two local *ready* and *go* phases are performed in parallel.

Similarly to ATOBar and ATOBarB, the Tree Topology Barrier (TTBar) and Tree Topology Barrier for Blades (TTBarB) implementations perform identically when the number of SPEs is lower than 9. Latency of TTBar and TTBarB increases afterwards due to the greater number of steps that are required to complete the *ready* and *go* phases. Observe, however, that they still scale much better than ATOBar. When more than 8 SPEs are involved, TTBarB avoids the scalability problems suffered by TTBar because the tree structure it implements for the *ready* and *go* phases reduces the number of inter-Cell signals, which finally translates into lower latency. Obviously, as the degree of the tree grows, performance of TTBarB progressively approaches to that of ATOBarB. For the sake of brevity, we present results just for degrees 2 and 3 though.

Finally, the Pairwise Exchange Barrier (PEBar) obtains the shortest latencies when the number of SPEs is a power of two because the number of steps is smaller than the tree topology algorithms. The extra steps required in the rest of the cases blur the advantages brought by the distributed way of synchronizing the SPEs that this algorithm entails. Note that all inter-chip signals are sent during the same phase with this algorithm which explains the almost constant latency increase when the number of SPEs is greater than 8.

4.2 Broadcast

Latency of the implementations of the broadcast primitive discussed in Subsection 3.2 is plotted in Figure 6 for 128-byte packets with 16 SPEs. Latency for larger packet sizes up to 16 KB takes the very same shape, although the absolute differences in terms of latency become more acute. Consequently, those results have been omitted for the sake of brevity.

As it can be seen in Figure 6, TTBCastBOpt obtains the best results in terms of latency when the number of SPEs is greater than 6, especially with a degree 2 tree. The tree structure assumed in TTBCastBOpt allows to initiate several DMA transfers simultaneously thus reducing the time employed in the *broadcast* phase. As expected, the larger the packet size, the lower the number of SPEs for

which TTBCastBOpt performs better than ATOBcast. For example, for 16-KB packets TTBCastBOpt is the best scheme when 3 or more SPEs are involved.

Additionally, Figure 7 shows the effective bandwidth (measured in GB/s) that is reached for the same implementations of the broadcast primitive as packet size varies from 128 bytes to 16 KB with 16 SPEs. As expected, absolute differences in terms of effective bandwidth grow with packet size. For example, a maximum bandwidth of 38.33 GB/s is reached with TTBCastB_{Opt} D2, whereas just 25 GB/s can be extracted from ATOBcastB_{Opt}.

4.3 Reduce

The latency of the reduce operations is shown in Figure 8. In all cases, results for reductions of 32-float vectors are presented. As already commented on, we strongly believe that this size is representative of most scientific applications, for which reductions involving simple operations on small data sizes are the prevalent case [12].

As it happens with barrier and broadcast primitives, the implementation of the reduce primitive with lowest latency depends on the total number of intervening SPEs. When the number of SPEs is less than 6, the simple ATORedB_{Opt} scheme proves to be the best option. However, as the number of SPEs increases TTRedB_{Opt} obtains lower latencies. When all SPEs are involved in the reduction, latency of ATORedB_{Opt} almost doubles the figures obtained with TTRedB_{Opt}. The fact that partial reductions distributed among all SPEs (instead of being centralized in a couple of root SPEs) are carried out in parallel is the key factor to explain such a latency decrease.

Apart from degree 2 trees, we have also evaluated a version of TTRedB_{Opt} with a degree 3 tree. However, as it can be observed in Figure 8, no latency advantage is found for TTRedB_{Opt} D3, which performs slightly worse than TTRedB_{Opt} D2 in all cases.

5 Conclusions

In this paper, we have described the design and implementation of three common collective operations: barrier, broadcast and reduce. For each of them we have explored several alternatives ranging from the naive approach to well-known algorithms coming from the cluster arena. Besides, we have adapted those in order to consider the architectural peculiarities and asymmetries of dual Cell-based blades. Our designs are efficient because, as we have explained, only require a signal register and a single 16 KB buffer. At the same time, our implementations of the algorithms are quite fast resulting in low latencies and high bandwidths for 16 SPEs. In addition, we have compared the performance of the different versions of each algorithm. This comparison highlights some interesting conclusions. On the one hand, the best algorithm is not always the same but depends on the number of SPEs and data size. On the other hand, the naive implementation obtains performance results close to the best algorithm in some cases.

In this way, experienced Cell BE programmers could opt for hybrid implementations that use the best algorithm depending on the input parameters. In the meantime, non-experienced programmers could adopt algorithms representing a tradeoff between performance and programming complexity.

References

1. Kahle, J., Day, M., Hofstee, H., Johns, C., Maeurer, T., Shippy, D.: Introduction to the Cell Multiprocessor. *IBM Journal of Research and Development* 49(4/5), 589–604 (2005)
2. Nanda, A., Moulis, J., Hanson, R., Goldrian, G., Day, M.N., D'Amora, B.D., Kessavarapu, S.: Cell/B.E. blades: Building blocks for Scalable, real-time, interactive, and digital media servers. *IBM Systems Journal* 51(5), 573–582 (2007)
3. O'Brien, K., O'Brien, K., Sura, Z., Chen, T., Zhang, T.: Supporting OpenMP on Cell. *International Journal of Parallel Programming* 36(3), 287–360 (2008)
4. Ohara, M., Inoue, H., Sohda, Y., Komatsu, H., Nakatani, T.: MPI microtask for programming the Cell Broadband EngineTM processors. *IBM Systems Journal* 45(1), 85–102 (2006)
5. Kumar, A., Senthilkumar, G., Krishna, M., Jayam, N., Baruah, P.K., Sharma, R., Srinivasan, A., Kapoor, S.: A Buffered-mode MPI Implementation for the Cell BETM Processor. In: 7th International Conference on Computational Science, Beijing, China (2007)
6. Bellens, P., Prez, J.M., Bada, R.M., Labarta, J.: CellSs: a Programming Model for the Cell BE Architecture. In: Proceedings of IEEE/ACM Conference on Super-Computing, Tampa, FL (2006)
7. McCool, M.D.: Data-Parallel Programming on the Cell BE and the GPU using the RapidMind Development Platform. In: Proceedings of GSPx Multicore Applications Conference, Santa Clara, CA (2006)
8. McCool, M.D.: Scalable Programming Models for Massively Multicore Processors. *Proceedings of the IEEE* 96(5), 816–831 (2008)
9. Abellán, J.L., Fernández, J., Acacio, M.E.: Characterizing the Basic Synchronization and Communication Operations in Dual Cell-Based Blades. In: 8th International Conference on Computational Science, Krakow, Poland (2008)
10. Yu, W., Buntinas, D., Graham, R.L., Panda, D.K.: Efficient and Scalable Barrier over Quadrics and Myrinet with a New NIC-based Collective Message Passing Protocol. In: Proceedings of Workshop on Communication Architecture for Clusters, Santa Fe, NM, USA (2004)
11. Velamati, M.K., Kumar, A., Jayam, N., Senthilkumar, G., Baruah, P., Sharma, R., Kapoor, S., Srinivasan, A.: Optimization of Collective Communication in Intra-Cell MPI. In: Proceedings of 14th International Conference on High Performance Computing, Goa, India (2007)
12. Petrini, F., Moody, A., Fernández, J., Frachtenberg, E., Panda, D.K.: NIC-based Reduction Algorithms for Large-Scale Clusters. *International Journal of High Performance Computing and Networking* 4(3–4), 122–136 (2005)
13. Kistler, M., Perrone, M., Petrini, F.: Cell Processor Interconnection Network: Built for Speed. *IEEE Micro*. 25(3), 2–15 (2006)
14. Hoefer, T., Mehlan, T., Mietke, F., Rehm, W.: A Survey of Barrier Algorithms for Coarse Grained Supercomputers. Technical report, Technical University of Chemnitz (2004)

Searching for Concurrent Design Patterns in Video Games

Micah J. Best¹, Alexandra Fedorova¹, Ryan Dickie¹, Andrea Tagliasacchi¹,
Alex Couture-Beil¹, Craig Mustard¹, Shane Mottishaw¹, Aron Brown¹,
Zhi Feng Huang¹, Xiaoyuan Xu¹, Nasser Ghazali¹, and Andrew Brownsword²

¹ Simon Fraser University

² Electronic Arts Blackbox

Abstract. The transition to multicore architectures has dramatically underscored the necessity for parallelism in software. In particular, while new gaming consoles are by and large multicore, most existing video game engines are essentially sequential and thus cannot easily take advantage of this hardware. In this paper we describe techniques derived from our experience parallelizing an open-source video game Cube 2. We analyze the structure and unique requirements of this complex application domain, drawing conclusions about parallelization tools and techniques applicable therein. Our experience and analysis convinced us that while existing tools and techniques can be used to solve parts of this problem, none of them constitutes a comprehensive solution. As a result we were inspired to design a new parallel programming environment (PPE) targeted specifically at video game engines and other complex soft real-time systems. The initial implementation of this PPE, Cascade, and its performance analysis are also presented.

1 Introduction

A video game engine is the core software component that provides the skeleton on which games are built and represents the majority of their computational complexity. Most game engines were originally written to be executed on machines with no facility for truly parallel execution. This has become a major problem in this performance hungry domain. In addition to the near ubiquity of multicore processors in consumer PCs the latest generation of gaming consoles have followed this trend as well. Microsoft's Xbox 360 and Sony's PlayStation 3 both feature multicore processors. To address this problem, video game engines are being restructured to take advantage of multiple cores.

Restructuring for parallelization has begun^[1], but much of the potential performance gains have yet to be realized as parallelizing a video game engine is a daunting task. Game engines are extremely complex systems consisting of multiple interacting modules that modify global shared state in non-trivial ways. This paper describes our experience of parallelizing a video game engine, our preliminary accomplishments, the lessons we learned and the insight for future research that emerged from this experience.

For our research, we use an open-source video game engine, Cube 2, which we enhanced by adding extra logic to AI and Physics modules so that it more closely resembles a commercial engine – we refer to this extended engine as Cube 2-ext. Careful inspection of this game engine, as well as our knowledge of commercial game engines, convinced us that hand-coding the engine to use threads and synchronization primitives would not only be difficult for an average programmer, but introduce a level of complexity that would limit even experts in extracting parallelism. Instead, we felt that relying on a parallel library that facilitates the expression of parallel patterns in sequential code and then parallelizes the code automatically would be more efficient. An evaluation of existing parallel libraries showed that none of them offered the support that we needed to express all the computation patterns present in game engines and so we created Cascade, our own library to fill this gap.

In its first incarnation, Cascade allowed a dependency-graph style of programming, where the computation is broken down into tasks organized in a graph according to their sequential dependencies and, most importantly, supported an efficient parallel implementation of a *producer/consumer* pattern. The producer/consumer pattern, pervasive in video games, consists of two tasks where one task (the producer) generates data for another (the consumer). While the pattern itself is not new, the parallel implementation and its semantic expression described in this paper are unique to Cascade. By applying this producer/consumer pattern we were able to achieve, using eight cores, a 51% decrease in the computation time necessary for the non-rendering or ‘simulation’ phase of Cube 2-ext.

While this first parallelization attempt was fruitful, we also learned about the limitations of the producer/consumer pattern and about the difficulty of trying to overcome these limitations in C++. In particular, we learned that whenever a computation had a *side-effect*, i.e., where the modification of some global state was required, the application of the producer/consumer pattern was very difficult without significant restructuring of the code. We also learned that the producer/consumer pattern did not allow for in-place transformations of data as a producer always generates a new copy of data. This was limiting in terms of performance and memory demands. At the same time, we were unable to find the required semantic constructs for expressing parallelism with side-effects and in-place transformations in existing C++ libraries. This has inspired us to design the Cascade Data Management Language (CDML). CDML, a work in progress, allows expression of parallel constructs and yet avoids these limitations. CDML is translated into C++, the language required by the game industry. We give a preliminary sketch of parts of CDML and describe its design philosophy.

In the rest of the paper we provide an overview of video game engines and the challenges involved in their parallelization (Section 2), we describe how we parallelized parts of Cube 2-ext using the new implementation of the producer/consumer pattern, briefly introduce Cascade and provide experimental analysis (Section 3). We describe the lessons learned and introduce CDML in Section 4. We describe related work and conclude in Sections 5 and 6.

2 Challenges in Parallelizing Video Game Engines

A game engine performs a repeated series of game state updates where a new state is generated for each video frame. Each part of this state is the responsibility of one or more distinct subsystem in the engine. The AI subsystem, for example, is responsible for dictating the behaviour of artificial agents in the game world while the Rendering subsystem is responsible for combining texture and geometry data and transferring it to the GPU. While the nature of data and computations in a particular subsystem can be quite different from another they are tightly coupled and the product of one subsystem may be required by several others.

A trivial attempt at parallelizing an engine could amount to running each of these subsystems in its own thread. This is far from ideal as the degree of parallelization would be limited by the number of subsystems and the differences between computational loads. Furthermore, this solution would be difficult to implement efficiently because the subsystems modify shared global state and interact in non-trivial ways. For instance, the AI subsystem updates the behaviours of the AI agents and then passes the control to Physics which simulates new positions and poses for characters' skeletons based on their behaviour, which are then used by the Renderer for display. This creates a series of *data dependencies* among these subsystems. If these subsystems run in parallel the global shared state must be protected, significantly limiting concurrency.

One alternative to using threads is to use parallel constructs such as parallel-for, found in OpenMP [2] and other libraries. Parallel-for is used for loop parallelization where the compiler relies on programmer-inserted directives to generate threaded code. While such tools are appropriate for some cases they are not sufficient to solve the entire problem. Data dependencies among different subsystems do not easily map to these simple parallel constructs making it difficult to express fine-grained and inter-subsystem parallelism. These approaches also exhibit a tendency for execution to alternate between periods of high parallelism and serial execution, leaving the serial phases as a critical bottleneck.

Furthermore, any parallelization tools must preserve the determinism of results and guarantee consistent performance to assure a seamless experience and correctness in multiplayer games. These constraints imply that using dynamic compilation or other kinds of dynamic optimizations are often a poor choice for this domain. In fact, some video game platforms simply forbid dynamic compilation.

In summary, video game engines are structured as multiple interacting subsystems with complex data dependency patterns making explicit threading and simple parallel constructs a poor option. A better solution would be to employ a PPE that allows the integration of diverse styles of parallelization methods.

3 Parallelizing Cube 2-ext

We identified the open source project Cube 2 as a test bed for our research. Despite having many attractive properties as an experimental test bed, Cube

2 lacked many modern game features such as fully realized physics simulation or character skinning. We addressed these limitations by adding some of these features to Cube 2. In particular, we added a more robust physics layer in the form of the popular Open Dynamics Engine (ODE) [3] and replaced the existing AI routines with more sophisticated versions that simulated ‘pack behaviour’. In the rest of the text we refer to this extended version as Cube 2-ext.

3.1 Parallelizing the Producer/Consumer Pattern

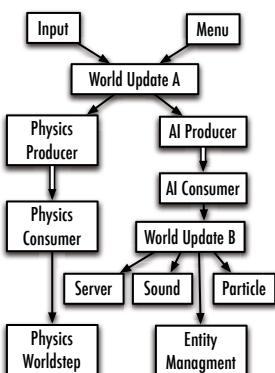
Our analysis of Cube 2-ext showed that in many places the computation was structured as a chain of interacting modules, or subsystems. Similar analysis applies to commercial video game engines with which we are familiar. For instance, the AI subsystem updates the behaviours of the AI agents and then calls Physics in order to assess the effect of the agent’s actions on the world. Further, within the Physics subsystem itself, the simulation first determines which game objects collide and then passes this data to the next stage which determines how to react to collisions.

We observed that a dependency-based decomposition of interrelated subsystems maps nicely to the well known dependency graph model [4][5][6][7]. In this model, the program is represented as a directed acyclic graph where each node represents a computational task and directed edges between nodes represent dependencies or data flow. We can execute tasks in parallel by mapping tasks to threads while respecting these dependencies. In our rendition of this model dependencies are one of two types, either logical or dataflow. In a logical dependency the parent must complete before the child commences execution. A dataflow dependency defines a relationship in which data comprising a shared state between parent and child is passed from the parent to the child when it is safe for the child to process it. Figure 1 shows the dependency graph of our preliminary parallelization of Cube 2-ext. In our graphical representation an edge from parent to child represents a dependency, hollow arrow denoting dataflow and thin arrow indicating logical dependency.

Fig. 1. The Cube 2-ext Dependency Graph

Additional parallelism is possible in the case of a dataflow dependency. If the parent has produced enough data that the child can begin processing the child can begin execution before the parent completes. We exploit this possibility when parallelizing with the producer/consumer pattern.

In Cube 2-ext we applied this pattern in the ODE Physics and AI subsystems. The Physics subsystem determines whether there is a collision between objects in the world, boxes bouncing off floors and walls in our case, and packs the resulting collision data in an array. This process comprises the *producer* part. Then, the *consumer* processes each object to determine how to respond to the collision.



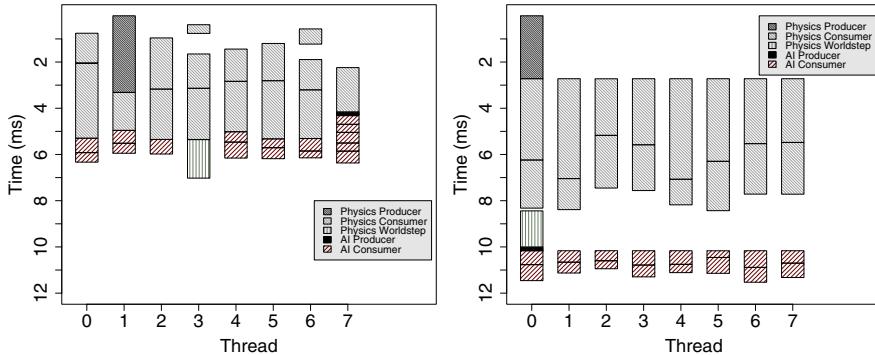


Fig. 2. Producers/consumers in Cube 2-ext(left) and Simulating parallel-for(right)

In the AI subsystem, the producer has been applied to the ‘monster control’ algorithm. The producer determines whether the monsters can ‘see’ each other and then makes this visibility data available to the consumer who processes it to determine the monster’s next action. Since the updates performed on each object in the consumer steps are independent the work of the task can be divided over multiple threads without synchronization.

The parts of a task that are mapped to threads at runtime are referred to as its instances. Note that although this producer/consumer pattern with multi-instance tasks is very similar to a data-parallel pattern using a parallel-for or parallel do-all, there is an important difference. In parallel-for or parallel do-all the parallel computation cannot begin before all the required data is ready as all parallel tasks commence simultaneously. However, with the producer/consumer pattern the first child instance begins as soon as the parent readies the first batch of data.

Figure 2(left) illustrates a typical execution for a single frame of the above task graph on 8 cores, showing the work done by each of the eight threads (x-axis) over time (y-axis). The AI and Physics represent the vast majority of calculations performed so we have eliminated the others from analysis for clarity. The experiments that follow were done on a Mac Pro system with two

```
// Create task objects.
// Task A: 1 instance
A a( 1 );
// Task B: batch size 2, 1 instance
B b( 2, 1 );
// Task C: batch size 1, 4 instances
C c( 1, 4 );
// Task D: batch size 1, 3 instances
D d( 1, 3 );
a->addDependent( b );
b->connectOutput( c );
c->connectOutput( d );
```

```
class C : public Task<int,int>
{
public:
    C(int batchSize, int numThreads) :
        Task<int,int>(batchSize, numThreads) {}

    void work_kernel(TaskInstance<int,int*>* tI) {
        for(int i=0;
            i < tI->receivedBatch->size();
            i++)
            tI->sendOutput(tI->receivedBatch->at(i)+1);
    }
};
```

Fig. 3. Declaring a task graph

Fig. 4. A task with work kernel

Quad-Core Intel Xeon CPUs. The $7ms$ execution time presented in this figure is typical based on averages of thousands of executions. Observe that the Physics consumer begins well before the producer has finished.

To demonstrate the difference between the producer/consumer pattern and parallel-for, we emulated parallel-for by adding restrictions to the task graph such that consumers did not begin processing data until the producer completed. Figure 2(right) demonstrates a typical execution of the above task graph on 8 cores when these restrictions are introduced. Notice that it takes $12ms$ to execute the task graph, longer than the $7ms$ in the unrestricted case.

3.2 Cascade

Although several existing PPEs compatible with C++ supported a dependency-graph pattern, such as Intel TBB [6], Cilk [4] and Microsoft PCP [7], none of them supported the concept of multi-instance tasks, which was key in our design. Dryad [5], a PPE that does support multi-instance tasks is targeted largely for database-style queries over computing clusters so it did not suit our needs either.

To fill this void we designed a new PPE, Cascade. Cascade began as a simple C++ PPE enabling the parallelization of producer/consumer and conventional dependency graph patterns with support for multi-instance tasks. In this section we describe the implementation of Cascade used for this paper and Section 4 details its future.

The key constructs in Cascade are *tasks*, *instances*, the *task graph* and the *task manager*. A Cascade task is an object that includes a work kernel and input/output channels. A work kernel is a function that encapsulates the computation performed by the task and an input or output channel is a memory buffer to facilitate communication. Tasks are organized into a computation graph according to their dependencies and the programmer specifies the producer/consumer semantics by connecting the tasks' input and output channels. Figure 5 shows an example dependency graph.

Figure 3 illustrates the C++ syntax used to map this dependency graph onto a Cascade graph abstraction. The programmer instantiates task objects, inheriting from one of Cascade's generic classes, specifying the number of instances to run in each task and the size of the batch that must be made ready by the parent before the child instance is launched. An example implementation of the class for task C from the example is shown in Figure 4.

As Figure 4 demonstrates, complexity of the code for creating parallel child instances is hidden from the programmer. The simplicity of semantics for creating producer/consumer relationships means that very few changes were required to the code in Cube 2-ext. Producer and consumer work functions were wrapped in work kernels and made to exchange data via input/output channels.

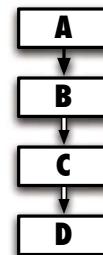


Fig. 5. Task graph from Figure 3

3.3 Cascade Performance Evaluation

For experiments in this section we used microbenchmarks, similar to that shown in Figure 5. The work kernel was almost identical to that in Figure 4, except for it was configured to perform some amount of ‘work’ after receiving the batch and before sending it out. The amount of work was varied depending on the experiment. In all cases, we report average performance based on at least 20,000 trials. The standard deviation in all cases was under 5% with the exception of experiments where the work size was very small, taking only a few microseconds to execute, or the number of threads was large, more than six – in these cases operating system activity, interrupts, and non-determinism in scheduling contributed to the noise.

3.4 Summary of Evaluation

In the first set of experiments we evaluated the scalability of Cascade. Figure 6 demonstrates the results. The x-axis shows the number of threads and the y-axis shows the time normalized to the single-threaded runtime. As the number of threads increases, we expect the running time to decrease. Scalability in Cascade may be limited due to contention on the *runqueue*, the queue of tasks whose dependencies have been satisfied and that are ready

to be assigned to a thread. The degree of runqueue contention depends on the amount of work done in the work kernel. Contention varies inversely proportional to the amount of work done. To demonstrate this effect, we show several curves for different work sizes, displayed in the number of instructions. With a small work size, 4000 instructions, the overhead of synchronizing on the runqueue hurts scalability and so beyond six threads we see no benefit. In fact, we witness performance degradation. With larger work sizes, however, the workload achieves nearly linear scalability, the running time with eight threads being 0.14 at 40K instructions. This demonstrates that the choice of work size is important for applying Cascade. In Cube 2-ext the work size in producers and consumers was on the order of millions of instructions.

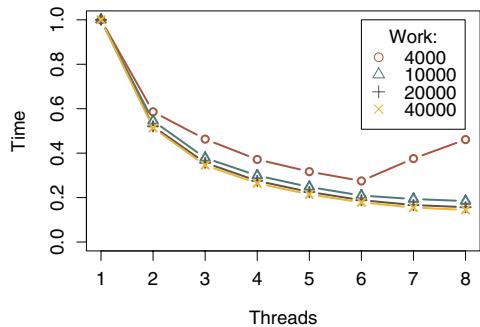


Fig. 6. Scalability with varying work

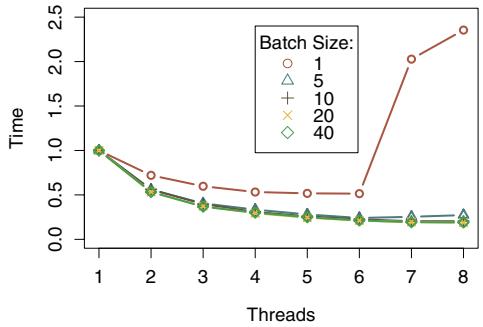


Fig. 7. Effect of batch size

The batch size is a measurement of the amount of data the producer accumulates before sending it to a consumer. We next examined the performance effects of varying this parameter. The smaller the batch the more work needs to be done to process it and the more synchronization there is on the batch queue. Figure 7 shows that with a very small batch size scalability is poor, but it quickly improves as the batch size is increased and we see reasonable scaling with batch sizes as small as 5 or 10.

Our analysis of Cascade demonstrates that scalability depends on the amount of work done by the task and the size of the batches. With work size of 10K instructions and a batch size of 10 scalability approaches linear. Using a lock-free queue as opposed to a mutex queue improves performance and scalability dramatically when the work size is small.

Applying the producer/consumer parallelization style to the Physics and AI subsystems of Cube2-ext reduced the average time per frame from 15.5ms using a single thread to 7.8ms using 8 threads. This gives the 51% speedup that we have claimed. Despite these improvements, there is still vast numbers of opportunities for parallelization that we have yet to exploit.

4 Cascade Data Management Language: Inspiration from the Lessons Learned

Our experience with Cascade and Cube 2-ext underscored the fact that there is no universal remedy that address all the complexities of this domain. Discussions from previous sections promote the dataflow approach expressed through the producer/consumer pattern as a good model for organizing programs and managing their interactions efficiently in parallel. While this model was effective for many cases, in our experiments we discovered that it has inherent limitations.

Problems parallelizing the AI subsystem illustrates these limitations. In this subsystem, each agent performs multiple checks to ascertain if another agent is visible. These tests were implemented using Cube's original routines to walk the octree structure that held the world data. In theory, walking an octree is a read-only algorithm, but as an optimization collision results were being cached in static buffers. Concurrent writes to these cache structures caused corruption and resulted in crashes. These unexpected side-effects were not readily apparent in the public interface and were further obscured by the complexity of the system. Efficiently managing this complexity in parallel is a primary motivation for Cascade.

This situation is a perfect example of how changes to the system's state are outside of the dataflow model. Transformations of the system state or *side effects* are not exceptions in the video game domain, but generally represent the only feasible solution to many problems. The problems with AI could have been prevented with a more sophisticated scheduling mechanism that is aware of side effects and schedules accordingly.

Our scheduler, built on dataflow principles, had only limited facility for incorporating state change information as dataflow is essentially stateless outside of

the contents of the communication channels. Logical dependencies serve as very coarse grained constraints, indicating that the state changes a parent makes must precede the child's execution. However, there is no direct association between a particular state change and the logical dependency and this dependency is only available between tasks and not between instances of the same task. Making specific information about these side effects available to the scheduler would allow it to correctly manage data accesses while promoting concurrency. Explicit and highly granular data dependencies allow for a number of optimizations particular to parallelizations such as an automatic Read-Copy Update [8] style solution to the multiple reader/one writer problem.

We have come to the conclusion that any effective parallelization strategy in this domain must provide rich and compact semantics for describing side effects in terms of transformations to the program state and provide first class support for specifying data dependencies. We believe that C++ is a poor choice to express these concepts, even mediated through a library. C++ pointers are of particular concern. Giving the programmer direct access to memory invalidates most guarantees of any system attempting to manage data. However, we acknowledge that C/C++ can be used to generate highly efficient serial code and is the industry standard language, especially in console development.

These observations have indicated the need for a system that requires the explicit expression of data interactions, contains primitives that reflect common data access patterns, allows for the decomposition of programs into dataflow tasks without an arduous amount of programmer effort and acknowledges the dominance of C++ in the domain. With this in mind we have begun work on CDML (*Cascade Data Management Language*) which addresses the problem of side effects inherent in procedural programming. State changes will be performed by composable transformations described in a declarative manner. Changes to system state are explicitly given as part of both the input and output to these function-like constructs. Larger units can be created by binding together these transformational constructs in a familiar procedural style. A program is created by joining these units together into an explicit dataflow graph. In this way all dependencies are expressible and many parallel optimizations are possible. This blending of styles allows the programmer to put emphasis on those best suited to the problem at hand.

CDML code will not actually be compiled, but instead translated to C++. To support the transition to CDML the embedding of C++ code will be supported. Unlike CDML, where stated dependencies are verified at translation time, the onus will fall on the programmer to ensure that all of the effects of C++ code on global state are declared.

While space limitations prevent us from giving a comprehensive example, we will illustrate how a programmer would take advantage of these concepts when implementing a single transformation. In AI systems it is common to pick a number of arbitrary members of a group as pack leaders. Expensive calculations, such as the line of sight testing in the AI subsystem discussed above, are performed only for pack leaders and the others inherit the results. In this case the player's

actions may attract the attention of monsters by causing some kind of disturbance. If a pack leader can see the player then that disturbance increases the alertness level of it and its followers. A list of any sufficiently affected monsters is sent to another part of the AI subsystem.

In order to implement this process, the programmer will need to iterate over a list of entities and apply a transformation to each one. The following CDML example provides the code for this transformation.

```

iterator distributeAlertness ( Entity[] enemies as agent )
  views: enemies, disturbanceLevel, playerPos < Input.check
  modifies: agent
  sends: hunters => AI.initiateHunt
{
  if ( agent.packLeader == true ) {
    if ( lineOfSight( playerPos, agent.position ) ) {
      agent.alertness += disturbanceLevel;
    }
  } else {
    agent.alertness = enemies[ agent.packLeader ].alertness;
  }
  if ( agent.alertness > HUNTINGTHRESHOLD ) {
    hunters << agent;
  }
}

```

CDML makes explicit the natural decision making process a programmer goes through when implementing an algorithm. First the scope of the transformation must be determined. Does it affect a single data item or collection? If the later, is only a subset affected or is the entire collection modified? Finally, is there an ordering to the modifications? In our example, we would like to process a collection structured as an index set, or array, and the ordering of updates is unimportant. The construct selected for this operation is the `iterator`, which corresponds in CDML to this data pattern. The signature of the transform defines the type accepted by this transformation (`Entity`) and defines a name (`agent`) for the individual element of the collection (`enemies`) being modified.

CDML will contain constructs adaptable to all the permutations of scope mentioned above and more. For example, if we only wanted to process a subset of the index set we could have specified a filter on the input. Filters are very important in CDML as they provide more information about the exact data affected by a transformation.

Following the signature is the *constraint block* which explicitly states the data dependencies of the transformation. For pure CDML code these dependencies are verified at translation time. There are several possible categories in the constraint block, but only `views`, `modifies` and `sends` are needed so the others are omitted.

The input required to perform the desired computation is specified in the `views` list. In this case we would like to read the other elements of the `enemies` array, the current `disturbanceLevel` and the players current position, `playerPos`.

In the case of `playerPos`, we would like to use it's value after it has been calculated by the input checking subsystem. Thus we use the '`<`' operator, to denote that we would like `playerPos` 'as modified by' `Input.check`. This is one way of specifying logical dependencies in CDML.

Output is either in the form of state changes as specified by the `modifies` list or directly output through named data channels as specified by the `sends` list. In the example, we would like to change the current agent's `alertness` value and produce a list of agents that reach the threshold to be further processed by `AI.initiateHunt`. Sending through a channel is done using the '<<' operator.

The information required for the constraint block represents considerations a programmer already makes. These characteristics of a process are simple for a programmer to specify, but can be extremely difficult to derive computationally. Most computations in this domain can be expressed as operations on structured collections and since constraints are inherited when transformations are composed, any given transformation will have a manageable number of constraints.

To further ease specification of data constraints, the constraint block will be refined in translation to eliminate, if possible, over-constraint. For example, the programmer could have added `enemies` to the `modifies` list, instead of `agent`. This would be detected and reduced to `agent` alone. The requirement to list all constraints in the code itself, as opposed to deriving them during translation, is partially to facilitate the embedding C++ and other languages and to encourage a parallelization-friendly programming style.

5 Related Work

Cascade shares many similarities with other PPEs such as Intel's TBB [6], Microsoft's Parallel Computing Platform (PCP) [7], and Cilk [4]. Those PPEs are also built on top of general-purpose languages (C++, C#, C) and also support parallelization of computations expressed as a dependency graph. Due to the reasons detailed in Section 3.2, we could not use any of these PPEs for our project. Dryad [5], a PPE designed largely for database-style queries over computing clusters, provides rich support for parallel patterns and does include functionality similar to multi-instance tasks. Although we have not used any of the existing PPEs in our work so far, we envision using them in our future work to parallelize concurrent patterns that are not supported in Cascade. OpenCL [9], a new open standard for C-based parallel environments supporting both CPU and GPU programming, seems well suited to this.

Cascade shares similarity with domain-specific PPEs, in that it was designed with application specific needs in mind. Thies et. al. described a library targeted for streaming applications [10]. Galois [11] is a library targeted at irregular data structures. While these domain-specific libraries target a particular computation pattern, Cascade has a broader focus in considering an *application* domain, and so it must support multiple computation patterns and consider domain-specific performance needs.

Finally, in our work we rely on research into algorithmic patterns. Although we do not cite all work in this area due to space limitations, we note a relatively recent report from UC Berkeley [12] that categorized computation patterns according to thirteen classes, or 'dwarfs'. Many of these 'dwarfs' are present in video game engines and so this classification system will be instructive in our further search for concurrent patterns in video games.

6 Summary

While parallelization of video game engines is a serious challenge faced by major video game companies [1], very little information describing details of this problem exists in the public domain. Our work sheds light on this subject and to the best of our knowledge this is the first detailed account of this problem in the public domain. We identified computations that could be parallelized by applying the producer/consumer pattern; this allowed us to achieve as much as 51% speedup on eight cores in AI and Physics subsystems. This process of parallelization and our analysis of existing tools informed the design of our PPE Cascade and more importantly shed light on the requirements for a set of tools that would truly encompass the patterns of this domain. Our experiences using Cascade showed that an even more general solution, one based around explicit data transformation and the expectation of side-effects is needed. To address this fact we are designing CDML, the Cascade Data Management Language, to leverage our constantly expanded and refined library and improve programmer performance while increasing program efficiency and correctness.

References

1. Leonard, T.: Dragged Kicking and Screaming: Source Multicore. In: Game Developers Conference (2007)
2. Chandra, R.: Parallel Programming in OpenMP. Morgan Kaufmann, San Francisco (2001)
3. Open Dynamics Engine, <http://www.ode.org/>
4. Blumofe, R.D., et al.: Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing* 37(1), 55–69 (1996)
5. Isard, M., et al.: Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.* 41(3), 59–72 (2007)
6. Reinders, J.: Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism. O'Reilly, Sebastopol (2007)
7. Microsoft Parallel Computing Platform,
<http://msdn.microsoft.com/en-ca/concurrency/>
8. McKenney, P.E., et al.: Read-copy update. In: Ottawa Linux Symposium, pp. 338–367 (2002)
9. Munshi, A.: OpenCL: Parallel Computing on the GPU and CPU (2008)
10. Thies, W., et al.: A practical approach to exploiting coarse-grained pipeline parallelism in C programs. In: MICRO (2007)
11. Kulkarni, M., et al.: Optimistic parallelism requires abstractions, pp. 211–222 (2007)
12. Asanovic, K.: et al: The Landscape of Parallel Computing Research: A View from Berkeley. Electrical Engineering and Computer Sciences, University of California at Berkeley, Technical Report No. UCB/EECS-2006-183, December 18(2006-183), 19 (2006)

Parallelization of a Video Segmentation Algorithm on CUDA-Enabled Graphics Processing Units

Juan Gómez-Luna¹, José María González-Linares², José Ignacio Benavides¹,
and Nicolás Gui²

¹ Computer Architecture and Electronics Department, University of Córdoba,
Campus de Rabanales, 14071 Córdoba, Spain
`{el1goluj, el1bebej}@uco.es`

² Computer Architecture Department, University of Málaga, Campus de Teatinos,
28071 Málaga, Spain
`{gonzalez, nico}@ac.uma.es`

Abstract. Nowadays, Graphics Processing Units (GPU) are emerging as SIMD coprocessors for general purpose computations, specially after the launch of nVIDIA CUDA. Since then, some libraries have been implemented for matrix computation and image processing. However, in real video applications some stages need irregular data distributions and the parallelism is not so inherent. This paper presents the parallelization of a video segmentation application on GPU hardware, which implements an algorithm for abrupt and gradual transitions detection. A critical part of the algorithm requires highly intensive computation for video frames features calculation. Results on three CUDA-enabled GPUs are encouraging, because of the significant speedup achieved. They are also compared with an OpenMP version of the algorithm, running on two platforms with multiples cores.

Keywords. Video Segmentation, CUDA, Canny, Generalized Hough Transform.

1 Introduction

A high number of video analysis applications, such as storyboard generation, video comparison, scene detection, video summarization, etc., apply temporal video segmentation as a previous step for further processing. Temporal video segmentation is a video processing technique, which is able to identify the shots appearing in a video, i.e., the sequences of frames captured from a single camera operation. Thus, a thorough study of video segmentation seems very convenient, in order to exploit their parallelism.

Two different types of transitions link the shots in a video. On the one hand, abrupt transitions or cuts occur when the change from one shot to the next one is performed in just one frame. On the other hand, gradual transitions involve several frames. Shot detection is performed through shot transition detection algorithms. This work focuses concretely on the on-line abrupt and gradual transitions detection algorithm presented in [1] and [2], which has achieved a high score in transitions detection.

The aforementioned applications are used in the digital video industry where, typically, cheap hardware computation platforms are available. Nowadays, these platforms may have a multi-core microprocessor, which is the trend in recent years. Applications should be parallelized in order to exploit the Thread Level Parallelism (TLP) inherent in these two- or four-core microprocessors. However, using the Graphics Processing Unit (GPU) as a SIMD coprocessor can also be considered.

Since the launch of nVIDIA CUDA [3], programming a GPU for general purpose computations is pretty much easier than before. The parallelizing effort with CUDA is equivalent to that with OpenMP [4] or any other application programming interface (API). Moreover, some libraries, such as OpenVidia [5] and GPUCV [6], have been implemented for matrix and image computations. However, the algorithm, scope of this work, is a typical video application, where some stages need irregular data distributions and the parallelism is not so inherent.

This paper presents the parallelization of the abrupt and gradual transitions detection algorithm, described in Section 2, on three CUDA-enabled GPUs: nVIDIA GeForce 8800 GTS, nVIDIA GeForce 9800 GX2, both with CUDA compute capability 1.1, and the recently released nVIDIA GeForce GTX 280, with CUDA compute capability 1.3. The performance of these implementations is compared with a data decomposition using OpenMP on a four-core Intel Core2Quad Q6600 2.40GHz and an eight-core Intel Xeon X5355 2.66 GHz multi-processor, in Section 5. Section 3 introduces CUDA and describes the decomposition of the algorithm in kernels. Section 4 explains the implementation of the most significant kernels.

2 Temporal Video Segmentation

This section presents a unified scheme for on-line video segmentation based on luminance and contour information from video sequences [1], [2]. It is able to detect abrupt as well as gradual shot transitions with high accuracy using neural networks for classification purposes. The process is summarized in Fig. 1.

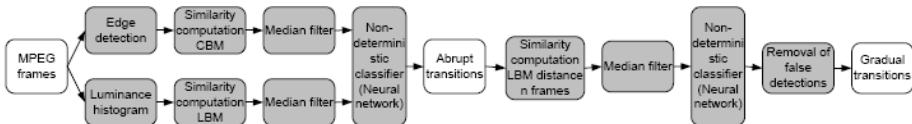


Fig. 1. Flow diagram to perform abrupt and gradual shot transitions detection. Shadowed stages indicate computing phases within the algorithm.

Two metrics are used to compare video frames, as it is explained below. The MPEG stream is decompressed and divided into frames, from which contour and luminance information is obtained:

- Contour information is extracted using the Canny edge detection algorithm [7]. Next, a Generalized Hough Transform (GHT, [8]) searches for couples of edge points whose gradient directions (θ) differ a certain angle (typically, two angles of 90° and 135° are used). When a pairing is found, a value α is computed. Then, the

(α, θ) element of a 2D histogram, called orientation table (OT), is updated ($OT(\alpha, \theta) = OT(\alpha, \theta) + 1$). The OT of an image can be used to detect if a given template, represented by its own OT, is present in the image. This algorithm uses a frame of the video as a template for the next frame. Thus, OTs of two frames are compared using a modified normalized 2D cross correlation [9] to obtain a similarity value (CBM, Contour Based Metric).

- A luminance histogram is computed for every frame. The histograms of two frames can be compared using a modified normalized 1D cross correlation function that obtains a similarity value (LBM, Luminance Based Metric).

Abrupt shot transitions are computed by comparing consecutive frames. The resulting similarity values of CBM and LBM are filtered by a third order median filter to minimize the effect of flashes and motion in the sequence. After that, a non-deterministic classifier identifies the abrupt transitions. The classifier is based on a multi-layer perceptron neural network with one hidden layer, 6 elements in the input layer, 10 neurons in the hidden layer, and 2 output elements.

Once abrupt transitions detection has been carried out, gradual shot transitions detection is performed by comparing frames at distance n (typically $n=15$). Candidate gradual transitions are identified by means of LBM over several frames and a new multi-layer perceptron neural network. This has 101 elements in the input layer, 80 neurons in the hidden layer, and 2 output elements.

Once candidate gradual transitions have been detected, it is required to discard false detections caused by camera operation. In order to do this, the algorithm implements a motion estimation procedure based on the GHT, which detects camera effects, such as zooms or displacements. Since displacements are the most usual effects, a Displacement Table DT is calculated for each frame. The displacement between a frame and the next ones in a window w (typically $w=10$) can be determined comparing the DT of each.

3 Implementing with CUDA

Since this algorithm is highly compute-intensive, its execution on a single CPU can take more than three seconds per frame. This value is too far from real-time, which would be a very desirable target for the applications it can be used. The scope of the algorithm is on-line video segmentation. Hence, parallelization should be applied to the computation of each frame. Some parts of this computation are really complex, such as the contour information extraction and the generation of the orientation table (OT). Moreover, some of them exhibit an inherent parallelism, because computation over all pixels of the image is regular. Thus, these parts present typical features of SIMD processing, what makes them suitable for parallelization on GPU. However, the parallelism is not so clear in other parts as explained next. Obtaining a performance improvement of these parts is the major challenge of this work.

3.1 CUDA Programming Interface

CUDA consists of a set of C language library functions and a compiler (nvcc), which generates the executable code for the GPU. CUDA offers a huge number of threads

running in parallel. Some elements should be described, because of their importance while programming:

- A **block** is a group of threads, which is mapped to a single multi-processor. All threads of a block can access 16 Kbytes of shared memory. The resources of the multi-processor are equally divided among the threads. The data is also divided among the threads in a SIMD fashion.
- The threads of a block run logically in parallel, but not always physically due to the limited hardware resources. A collection of threads running concurrently at the same time is called a **warp**. The size of a warp is 32.
- Every thread executes the same code called **kernel**. The number of blocks and the number of threads in a block should be defined for each kernel.

From the hardware point of view, threads execute on SIMD single multi-processors. The number of multi-processors depends on GPU model, varying from 1 in the first CUDA-capable models to 30 in the GeForce GTX 280.

All the threads of a kernel access the global memory or device memory. Data is transferred to global memory from the memory of the host, that is, the CPU. Moreover, each multi-processor contains a number of registers, which are split among the threads of a block.

The maximum efficiency of a kernel can be obtained when the occupancy of the multi-processors is as high as possible and the memory hierarchy is properly used.

On the one hand, in order to determine the occupancy, the code is compiled using the nvcc and a special flag that outputs the amount of memory and the registers consumed by the kernel. Analyzing these values permits to determine the number of threads of a block for maximum efficiency. Usually, a block should contain 128-256 threads to minimize execution time. The number of blocks is set according to entry data size. On the other hand, global memory is a high-latency memory. Its bandwidth is used most efficiently when simultaneous memory accesses by threads can be *coalesced* into a single memory transaction. Coalescing occurs when the threads access words in sequence, i.e., the k^{th} thread access the k^{th} word, and the words lie in the same segment of 32, 64 or 128 bytes. Furthermore, a proper use of the low-latency shared memory can also improve performance. Threads of a block access shared memory simultaneously when the data resides in different memory banks. If it does not, simultaneous accesses are serialized.

3.2 Decomposition of the Algorithm in Kernels

This implementation with CUDA is focused on obtaining the similarity values of CBM and LBM, which are the most time-consuming stages of the algorithm. The execution time of the classifiers and the motion estimation is negligible. Thus, they are delegated to the CPU.

Both metrics are implemented with several kernels. Defining what tasks are coded into a kernel is a programmer's decision. In this work, most of the kernels are oriented to computation, but others prepare data for further processing by other kernel. Moreover,

some kernels are based on sample codes, included in the CUDA SDK, which implement commonly used functions, such as separable convolution [10] or histogram [11].

CBM and LBM require, first, a features extraction and, second, the comparison of these features. CBM is divided into edge detection, orientation table generation and correlation between consecutive orientation tables. Features extraction in LBM consists of generating a luminance histogram for each frame. After that, histograms are correlated using a 1D cross correlation. Table 1 presents the kernels of this implementation.

Table 1. Collection of kernels used in the implementation of CBM and LBM. Some of them are custom developed, while others are based on CUDA separable convolution and histogram.

Stage	Kernel	Description	Development
Canny Edge Detection	Convolution Row	Row filtering	Based on CUDA sample
	Convolution Column	Column filtering	Based on CUDA sample
	Gradient Calculation	Gradient computation	Custom
	Non-Maximum Suppression	Determines if a pixel belongs to a contour	Custom
Generalized Hough Transform	Compact List	Compacts sparse contour points matrix into a dense list	Custom
	OT Generation	Search pairings for each contour point. Generates the orientation table	Custom
2D Cross Correlation	Table of Terms in a Window	Adds an element of the OT and the elements in a square window	Based on CUDA sample
	Normalization Term	Generates the normalization term of an OT	Custom
	1D Correlation	Correlates an OT and a table of terms in a window. The result is the contour similarity value	Custom
Luminance Histogram	64 bins Histogram	Generates a 64 bins histogram of an image. Per-thread sub-histograms are reduced with global memory atomic functions	Based on CUDA sample
1D Cross Correlation	Vector of Terms in a Window	Generates a vector of terms in a window. Implemented by a row convolution with a 1D filter of size 3 elements equal to 1	Based on CUDA sample
	Normalization Term	Normalization term of a histogram	Custom
	1D Correlation	Correlates a histogram and a vector of terms in a window of the next frame. The result is the luminance similarity value	Custom

4 Parallelizing the Contour Based Metric

CBM requires much more execution time than LBM. In fact, the generation of the OT is the most time consuming step. Moreover, once the edge detection has been performed, computation turns irregular. The number of contour points obviously depends on the frame. Thus, the generation of the OT is workload dependent. The challenge of this work is finding an efficient implementation, which guarantees a constant performance improvement.

4.1 Canny Edge Detection

This work implements a version of the Canny algorithm without thresholding with hysteresis, which is usually its slowest part. This version of the Canny algorithm is adapted to the abrupt and gradual transitions detection algorithm, scope of this work, and is not an entire implementation as is the recently presented in [12].

The first part of the Canny algorithm convolves the image with two Gaussian filters, in order to reduce noise. Both filters are separable, what makes easier and more efficient its implementation. This is based on a separable convolution kernel included in the CUDA SDK. The second part obtains the intensity and direction of the gradient. This computation is applied over all pixels of the image. Finally, the last part is carried out to determine if the gradient magnitude assumes a local maximum in the gradient direction. A pixel belongs to a contour, if its gradient magnitude is greater than a threshold and the gradient magnitudes of the adjacent pixel in the gradient direction. Fig. 2 explains this process.

Convolution separable and gradient calculation are regular as they are applied to all pixels in the same way. Hence, their adaptation to GPU is easy and achieves an important improvement. The last part, non-maximum suppression, presents some undesirable features for GPU, because computation depends on the gradient direction. This is implemented with conditional clauses, which involve different alternatives for the threads in a warp. Thus, some threads in the warp are not executed simultaneously, i.e. thread execution is serialized. Performance can be improved by loading data, which is being reused several times during the kernel, into shared memory. Each thread block loads a row tile, the upper row tile and the lower row tile into its shared memory, as is shown in Fig. 2.

4.2 Generalized Hough Transform

As it has been seen, Canny algorithm returns a sparse matrix of contour points. After that, computation turns very irregular along the pixels of the image, turning the parallelization on GPU pretty much difficult.

A straightforward implementation can use one thread per pixel. If the pixel is a contour point, the thread will search the rest of contour points, in order to check whether there is a pairing or not. This approach does not achieve a good performance, since it requires conditional clauses, which prevent the threads in a warp from executing simultaneously. Thus, computation will be serialized and the computing power of the GPU, wasted. In fact, most of the threads will turn idle, because only a small amount of threads correspond to a contour point. Moreover, these active threads should examine the whole image, in order to find a reduced number of contour points.

A better implementation should reorganize the input data, in order to maintain every thread active. In this paper, contour points are compacted and then assigned to the threads. Thus, the generation of the OT consists of two kernels.

Compacting the Contour Points. The first kernel compacts the contour points in a single dense list, which contains the coordinates of the contour points and their gradient directions. Each thread evaluates one pixel. If it is a contour point, the thread increments an accumulator, lied in global memory, using an atomic addition. The

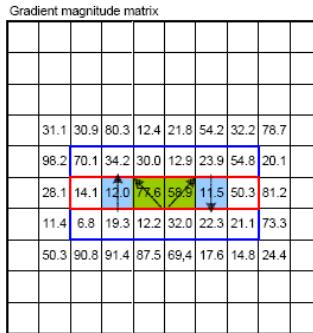


Fig. 2. Non-maximum suppression. A thread block is focused on a row tile (red), but also loads to shared memory the upper and the lower row tiles (blue). A pixel belongs to a contour if its gradient magnitude is greater in the direction of the gradient (lime). If it is not, the pixel is discarded as contour point (pale blue).

current value of the accumulator is used as index for the compact list, as is showed in Fig. 3. Thus, the size of this list is the number of contour points, which is dependent on the frame. This conversion from sparse matrix to dense list makes more efficient the subsequent data management and computation.

Generating the Orientation Table. The compact list is the input data to the second kernel. It uses a number of thread blocks, which depends on the number of contour points. While the algorithm is processing a video, the size of the thread block is fixed. Instead, the number of blocks is varying in each call to the kernel, because it is the number of contour point divided by the size of the thread block. This makes the kernel adaptable to any image. Each thread block takes a part of the compact list and loads it in shared memory. Then, it searches pairings for each contour point, comparing the gradients within its own sub-list during the first iteration. Next iterations require loading in shared memory the sub-list correspondent to other blocks, in order to compare the gradient of a contour point with any other in the list. Fig. 4 explains this process. Each time a pairing is found, the corresponding element of the OT, which resides in global memory, is incremented by using an atomic addition. Thus, the OT is generated as a square histogram.

An alternative to the use of atomic additions, which unavoidably serialize the execution, could be generating an OT per thread in shared memory. Each thread would be assigned to one contour point and would search pairings for it, in order to create its own OT. Finally, all OTs would be added by a reduction. The major drawback of this strategy is that the size of the OT is 45x45. Thus, when more than two threads are assigned to a block (typically, the number of threads per block is 64) the private OTs do not fit into shared memory and these tables need to be mapped into global memory. Taking into account that the accumulation operation generates non-coalesced memory access pattern, the high latency of the global memory will not allow to obtain good performance values.

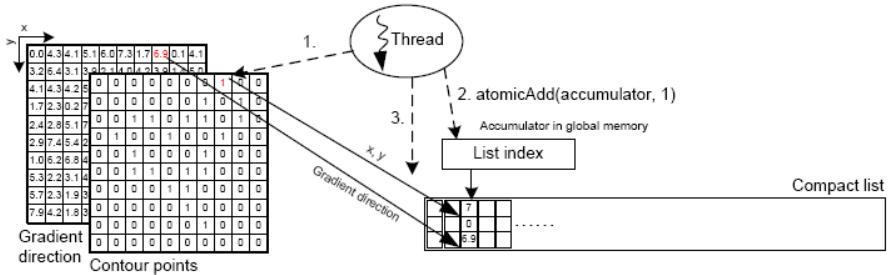


Fig. 3. Sparse matrix to dense list conversion: 1. The thread checks if the pixel is a contour point; 2. If it is, the thread increments an accumulator in global memory; 3. Copies the contour point coordinates and its gradient direction into the compact list.

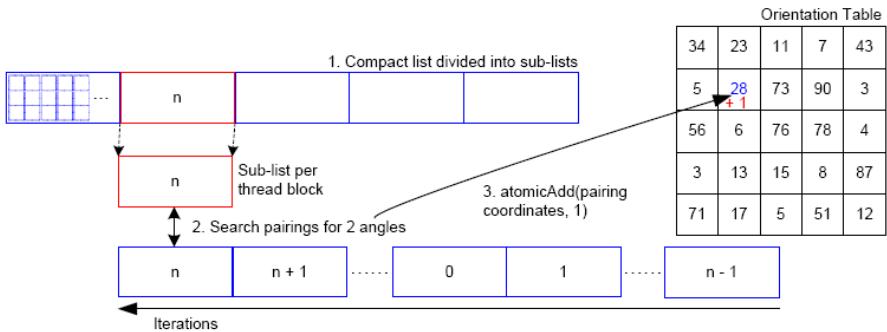


Fig. 4. OT generation: 1. Compact list is divided into sub-lists and these are assigned to each thread block; 2. The block search pairings in all the sub-lists; 3. If a pairing is found, the corresponding element of the OT in global memory is incremented using an atomic addition.

4.3 2D Cross Correlation

The preceding stage calculates the OT of an image, which is a dense matrix. Computation is regular again. During the first part of this stage, each element of the OT is windowed, i.e. it is added to the elements of a square window centered in it. This generates a table of terms in a window of the same size than the OT. This part can also be implemented using the separable convolution and two one-dimensional filters with all the elements equal to 1. The table of terms is a window of a frame is then correlated with the OT of the preceding frame and the same OT rotated clockwise and counterclockwise, resulting three similarity values. The highest of these values is the result of the kernel.

5 Experimental Results

This section presents the evaluation of the performance of the implemented algorithm. First, results on three CUDA-enables GPUs are analyzed. Then, the performance of a multi-core implementation with OpenMP on two platforms is showed, in order to compare with the results on the GPUs.

5.1 Performance on CUDA-Enabled GPUs

This implementation has been tested on three CUDA-enabled GPUs, whose features are showed in Table 2. They permit to compare the performance of the algorithm on the three more recent nVIDIA GPU generations. As it can be seen, the major difference between GeForce 8 and GeForce 9 series is the number of multiprocessors. Both 8800 GTS and 9800 GX2 have compute capability 1.1. This permits global atomic functions on 32-bit words in global memory. GeForce 200 series has compute capability 1.3, which improves features significantly. It allows atomic functions in global and in shared memory and supports double-precision floating-point numbers.

The workloads for the tests are 500 frames fragments of four MPEG videos from the MPEG-7 Content Set. Table 3 shows the average number of contour points and pairings per frame. These values have a clear influence on execution time, as it is explained below. Fig. 5 presents the execution time for the four fragments of the videos. GeForce GTX 280 improves significantly the performance of its predecessors. The main reason is the greater number of multi-processors. GeForce 9800 GX2 does not outperform significantly the results of GeForce 8800 GTS, since only one core of 16 multi-processors is used.

Table 2. Hardware and software features in nVIDIA GeForce GPUs

Parameter	8800 GTS	9800 GX2	GTX 280
Multi-Processors per GPU	12	2x16	30
Processors/Multi-processor	8	8	8
Threads/Warp	32	32	32
Threads/Block	512	512	512
Threads/Multi-Processor	768	1024	1024
Warps/Multi-procesor	24	24	32
32-bit registers/Multi-Processor	8192	8192	16384
Shared memory/Multi-Processor	16 Kbytes	16 Kbytes	16 Kbytes
Global memory	512 Mbytes	1 Gbyte	1 Gbyte

A previous analysis of the execution time of each kernel concluded that the generation of the Orientation Tables takes more than 90% of the execution time of the algorithm. This is clearly conditioned by the number of contour points and pairings. This is the main bottleneck for achieving a better performance in this application. The reason is the high latency of the global memory, which is accessed continuously in order to load sub-lists into shared memory, while searching for pairings. Moreover, once a pairing has been found, the corresponding element of the OT is incremented by using an atomic addition, which serializes the work of the threads. Although other parts of the algorithm, such as Canny, achieve a speedup of 15, the global improvement is limited by the pretty much slower OT generation.

In line with the efforts of accelerating the algorithm, *stream management* capability of CUDA has been used in order to overlap the transference of the frames from host memory to device memory. Unfortunately, this transference is less than 1% of the execution time of the kernels. Thus, the improvement is negligible.

Table 3. Test workloads characteristics. The four videos have a resolution of 352x288 pixels. Number of contour points and pairings are average values per frame, in the 500 frames fragments.

Video	Description	Contour points	Pairings
Basket	Basketball match: Images of the court and people in the stands	24902	22348724
Cycling	Cycling race: A cyclist on the road, surrounded by public	10850	5354556
Drama	Television series: Some actors in an indoor scenario	21714	19167915
Movie	Beginning of a movie: Credits on a black background	2833	615818

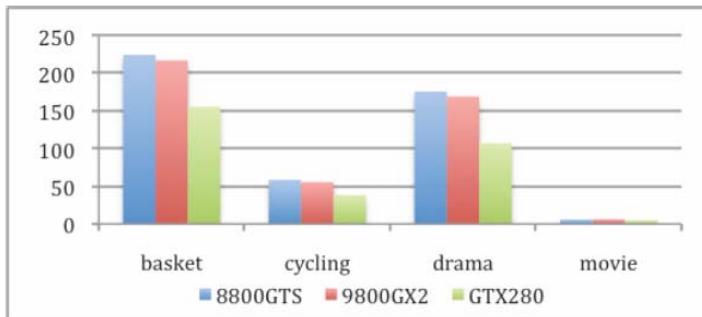


Fig. 5. Execution times of the CUDA implementation on the GPUs for the four videos. Results are represented in seconds.

5.2 An OpenMP Implementation

The abrupt and gradual transitions detection algorithm has also been implemented with OpenMP, in order to compare the performance on the GPUs and on two multiple cores platforms. Table 4 shows some features of both platforms.

The OpenMP implementation takes advantage of the Thread Level Parallelism in both platforms by following a data decomposition strategy. Parallel-for pragmas are used to assign equally the workload to the threads. Fig. 6 presents the results on both platforms for the videos in Table 3. As in the CUDA implementation, the execution time is conditioned by the number of contour points and pairings. Speedup, referred to the 1 thread version, varies in a similar way in both platforms, as shown in Table 5. It scales well for 2 and 4 threads, but falls when the contour points and the pairings decrease or when the number of threads increases, that is, the implementation performs worse when the workload per thread decreases.

Table 4. Hardware features of both multiple core platforms

Platform	8-way Intel Xeon	Intel Core2Quad
# Sockets	2	1
Cores/Socket	4	4
Clock speed	2.66 GHz	2.40 GHz
L2 Cache	4 Mbytes/2 cores	4 Mbytes/2 cores

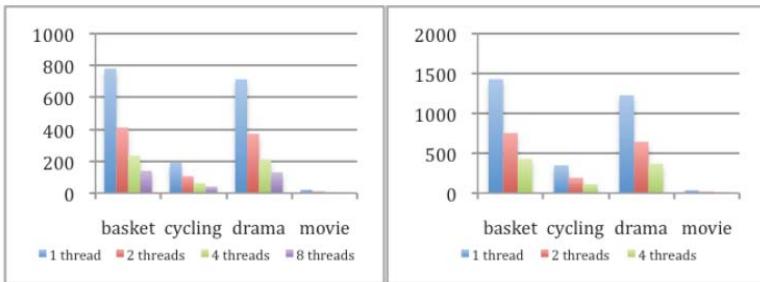


Fig. 6. Execution times of the OpenMP implementation on 8-way Intel Xeon (left) and on Intel Core2Quad (right) for the four videos. Results are represented in seconds.

Table 5. Speedup of the OpenMP implementation for both platforms

Platform	Video	2 threads	4 threads	8 threads
8-way Intel Xeon	Basket	1.90	3.32	5.57
	Cycling	1.81	3.07	4.65
	Drama	1.92	3.37	5.47
	Movie	1.83	2.54	3.61
Intel Core2Quad	Basket	1.90	3.32	-
	Cycling	1.81	3.09	-
	Drama	1.90	3.32	-
	Movie	1.83	2.53	-

5.3 Comparison between CUDA and OpenMP Implementations

The main goal of this work is the implementation of the algorithm in CUDA, but it also has been developed an implementation in OpenMP, in order to compare their performances. If the GPUs are being used as SIMD coprocessors, they should improve the results of current multi-core CPUs. In this way, if the results on the GeForce GTX 280 are compared with the ones on Intel Core2Quad and Intel Xeon, the conclusions are very encouraging.

Performance on GeForce GTX 280 achieves a speedup from 7.6 to 11.3 versus the 1 thread implementation on Intel Core2Quad, depending on the video. Moreover, the higher speedup corresponds to a greater number of pairings in the frames. Comparing with the 4 threads implementation on the same CPU, the speedup varies between 2.8 and 3.5. This is almost four times faster than one of the most recently released desktop processors.

Only the 8 threads implementation on Intel Xeon is in the same order of magnitude than GeForce GTX 280. The 8 threads implementation slightly beats the GTX 280 results for the basket video, but not for the rest. The GTX 280 improves up to 22% the execution time of the 8 threads implementation for drama and movie.

6 Conclusions

This paper has presented the parallelization of an abrupt and gradual transitions detection algorithm on CUDA-enabled GPUs. The implementation of some stages with

irregular computation and the conversion between storing formats, i.e., sparse matrix to dense list, have been the major challenge. Its evaluation on the three more recent nVIDIA GPU generations has revealed a very good performance and has shown the possibility of using GPUs for accelerating video segmentation applications.

The implementation on the recently released nVIDIA GeForce GTX 280 achieves an acceleration up to 11.3 comparing to a 1 thread implementation of the algorithm on a current desktop processor. Moreover, it is up to 3.5 times faster than a 4 thread OpenMP implementation, which has also been developed. Only an implementation on an 8-way multi-processor approaches the performance on GeForce GTX 280. This is a sample of the computing power of the GPUs and their applicability for general-purpose processing. Its use is also clearly justified in terms of costs, since the price of a current GPU is 10% of an 8-core Intel Xeon.

References

1. Sáez, E., Benavides, J.I., Guil, N.: Reliable Real Time Scene Change Detection in MPEG Compressed Video. In: IEEE International Conference on Multimedia and Expo. (2004)
2. Sáez, E., Palomares, J.M., Benavides, J.I., Guil, N.: Global Motion Estimation Algorithm for Video Segmentation. In: IS&T/SPIE Visual Communications and Image Processing (2003)
3. Compute Unified Device Architecture (CUDA), <http://www.nvidia.com/cuda>
4. Open SMP Programming (OpenMP), <http://www.openmp.org>
5. OpenVIDIA: Parallel GPU Computer Vision,
<http://openvidia.sourceforge.net>
6. Farrugia, J.P., Horain, P., Gueheneux, E., Alusse, Y.: GPUCV: A Framework for Image Processing Acceleration with Graphics Processors. In: IEEE International Conference on Multimedia and Expo. (2006)
7. Canny, J.F.: A Computational Approach to Edge Detection. IEEE Trans Pattern Analysis and Machine Intelligence 8, 679–698 (1986)
8. Guil, N., González, J.M., Zapata, E.L.: Bidimensional Shape Detection using an Invariant Approach. Pattern Recognition 32, 1025–1038 (1999)
9. Sáez, E., González, J.M., Palomares, J.M., Benavides, J.I., Guil, N.: New Edge-Based Feature Extraction Algorithm for Video Segmentation. In: IS&T/SPIE Symposium, Image and Video Communications and Processing (2003)
10. Podlozhnyuk, V.: Image Convolution with CUDA. nVIDIA white paper (2007)
11. Podlozhnyuk, V.: Histogram Calculation in CUDA. nVIDIA white paper (2007)
12. Luo, Y., Duraiswami, R.: Canny Edge Detection on NVIDIA CUDA. In: IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops (2008)

A Parallel Point Matching Algorithm for Landmark Based Image Registration Using Multicore Platform

Lin Yang^{1,3}, Leiguang Gong², Hong Zhang¹,
John L. Nosher³, and David J. Foran^{1,3}

¹ Center for Biomedical Imaging & Informatics, The Cancer Institute of New Jersey, UMDNJ-Robert Wood Johnson Medical School, Piscataway, NJ, 08854, USA

² IBM T. J. Watson Research, Hawthorne, NY, 10532, USA

³ Dept. of Radiology, UMDNJ-Robert Wood Johnson Medical School, Piscataway, NJ, 08854, USA

Abstract. Point matching is crucial for many computer vision applications. Establishing the correspondence between a large number of data points is a computationally intensive process. Some point matching related applications, such as medical image registration, require real time or near real time performance if applied to critical clinical applications like image assisted surgery. In this paper, we report a new multicore platform based parallel algorithm for fast point matching in the context of landmark based medical image registration. We introduced a non-regular data partition algorithm which utilizes the K -means clustering algorithm to group the landmarks based on the number of available processing cores, which optimize the memory usage and data transfer. We have tested our method using the IBM Cell Broadband Engine (Cell/B.E.) platform. The results demonstrated a significant speed up over its sequential implementation. The proposed data partition and parallelization algorithm, though tested only on one multicore platform, is generic by its design. Therefore the parallel algorithm can be extended to other computing platforms, as well as other point matching related applications.

1 Introduction

Point matching is crucial for many computer vision and image analysis applications, such as medical image registration. It is a computationally intensive process due to the calculation of the correspondence among a large number of data points. However, the medical image registration, when used for critical applications for instance the image assisted surgery, often requires real time or near real time performance. A lot of efforts have been made to speed up the point matching method and its related application such as medical image registration. To our knowledge, there has no reports about parallelization of the landmark based image registration algorithm on the multicore platform, the IBM Cell Broadband Engine. As we will show the proposed algorithm indeed contains a distinctive advantage for parallelization by its design.

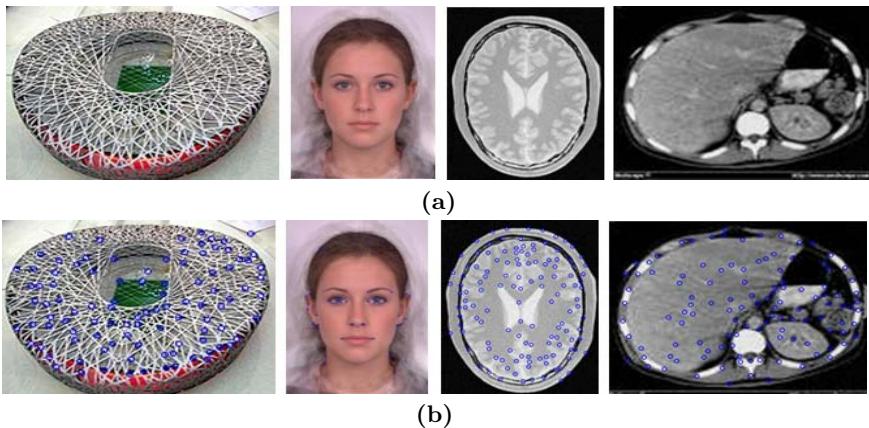


Fig. 1. The landmark detection using Harris corner detector. (a) The original images. (b) The images with the landmarks overlayed.

Image registration is the process to determine the linear/nonlinear mapping between two images of the same object or similar objects acquired at different time, or from different perspectives. Image registration is widely used in remote sensing, image fusion, image mosaic and especially in medical image analysis. It is very common that the images taken under different acquisition conditions, such as shifting of the patient's positions or changing of the acquisition parameters, show different appearance. Image registration can be separated as rigid registration [12] and non-rigid registration [34]. Given two images taken at different time (usually one is called fixed image and the other is moving images), the problem can be described as finding a linear/nonlinear transformation which maps each point in the fixed image to a point in the moving image. For nonlinear registration, some of them describe the transformation based on elastic deformations, such as fluid deformation based algorithm [56] and Demon's algorithm [78]. The others model the transformation by a function with some parameters, such as B-spline based image registration algorithm [9].

Nonrigid image registration in general is computationally expensive. With the advent of unprecedented powerful computing hardware, it becomes possible for fast nonlinear image registration using the multicore platform. In this paper, we will introduce a parallelization of a fast and robust image registration algorithm using a multicore processor platform, the IBM Cell/B.E. The algorithm starts by automatically detecting the landmarks in the fixed image followed by a coarse-to-fine estimation of the linear and nonlinear mapping. A landmark point is characterized by an affine invariant local descriptor, the multi-resolution orientation histograms. The corresponding landmarks in the moving images are identified by matching the local descriptors of candidate points. The point matching procedure is the most time-consuming therefore was accelerated by the

proposed parallelization algorithm on the IBM Cell/B.E. RANdom SAmple Consensus (RANSAC) [10] is used as a robust estimator to reject outliers in the corresponding landmarks. The final refined inliers are used to estimate a Thin Spline Transform (TPS) [11] to complete the final nonlinear image registration. The algorithm is completely unsupervised and computationally efficient. We have tested the method with several different types of images. The experimental results have shown significant speed up over the sequential implementation. The parallelized algorithm has the ability to handle very large transformations and deformations, while still providing accurate registration results. The proposed image registration algorithm is explained in Section 2. In Section 3, we describe the parallelization implementation using a IBM Cell Broadband Engine (Cell B./E.) multi-core processor. The experimental results are shown in Section 4. We briefly survey the related work in Section 5. Section 6 concludes the paper.

2 Landmark Based Image Registration

The proposed nonlinear image registration algorithm can handle large scale of transformations. It begins by automatically detecting a set of landmarks in both fixed and moving images, followed by a coarse-to-fine estimation of the nonlinear mapping using the landmarks. Robust estimation is used to find the robust correspondence between the landmarks in the fixed and moving image. The refined inliers are used to estimate a nonlinear transformation T and also wrap the moving image to the fixed image.

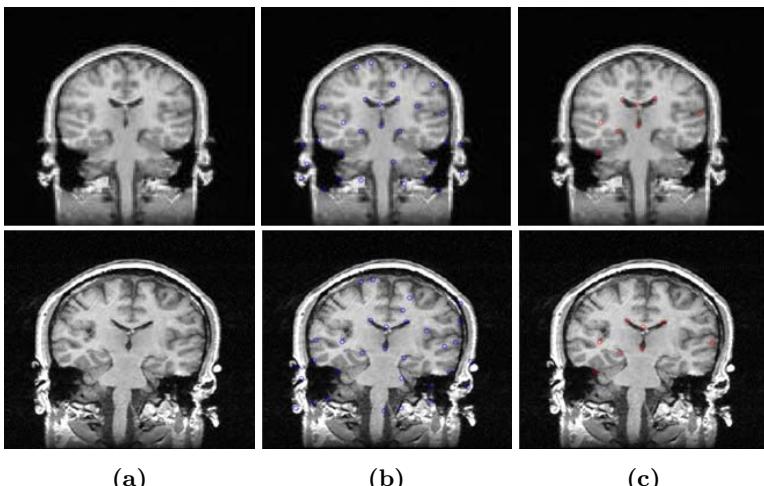


Fig. 2. Apply the robust estimation to find the robust landmark correspondence. (a)The original fixed and moving image. (b)The original pairs of matching points. (c)The robust matching points after rejecting the outliers.

2.1 Landmark Detection

The automatic landmark detection is the procedure used to accurately detect the prominent and salient points in the image. Harris corner detector was applied to find the points with the large gradients in both directions (x and y for 2D images). The original computation in the Harris corner detector involves the computation of eigenvalues. Instead, the determinant and trace are used in our algorithm to find the corners where α is chosen as 0.1.

$$F = \det(A) - \alpha \text{trace}(A) \quad (1)$$

Some representative landmark detection examples are shown in Figure 11.

2.2 Landmark Point Matching

After we detect the landmarks, we can extract features from the neighborhood of each landmark. The local orientation histograms are used as the features for landmark matching. The image is first convolved with the orientation filters. The filtering response in the neighborhood around the landmarks is computed to form the orientation histogram. Let $G_x(i, j)$ and $G_y(i, j)$ represent the gradients on pixel $p(i, j)$ along x and y direction, respectively. The orientation histogram h_k is defined as

$$h_k(i, j) = \begin{cases} C(i, j), & \theta(i, j) \in \text{bin}_k \\ 0, & \text{otherwise} \end{cases} \quad (2)$$

where

$$C(i, j) = \sqrt{G_x(i, j) + G_y(i, j)} \quad (3)$$

and

$$\theta(i, j) = \arctan \left(\frac{G_y(i, j)}{G_x(i, j)} \right). \quad (4)$$

The orientation histogram encodes the directions of the edges at each landmark point. It is proven to be an effective feature descriptor when the training samples were small [12].

In order to achieve robust matching of the landmarks, the extensive searching in the image space is required. This step is time consuming and often create the bottleneck for the landmark based image registration algorithm. In Section 2.4, we will show the time profile for each step in the whole procedure and it will be clear that the landmark matching dominate the execution speed.

2.3 Robust Estimation and Nonlinear Image Registration

Because the original matching landmark sets contain missing landmarks. RANdom SAmple Consensus (RANSAC) [10] is used to reject outliers and robustly estimate the transformation. The RANSAC robust estimator randomly selects

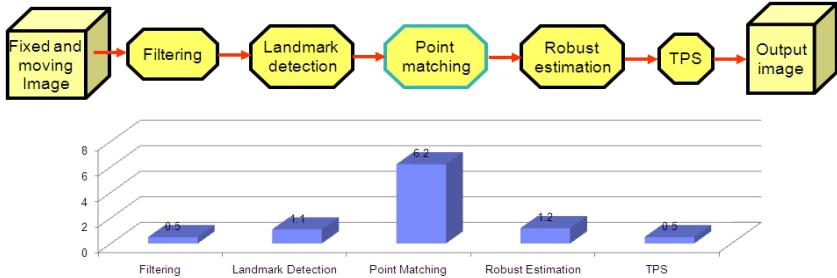


Fig. 3. The time profile for the image registration algorithm

the minimal subset of the landmarks to fit the model. Measured by a cost function, the points within a small distance are considered as a consensus set. The size of the consensus set is called the model support M . The algorithm is repeated multiple times and the model with largest support is called the robust fit. In Figure 2 we show the results of applying robust estimation to reject the outliers in the original matching landmarks. The Harris corner detector detected 32 landmark pairs in Figure 2b. Based on the assumption of an affine transformation, the RANSAC found 8 inliers (shown in Figure 2c) and the rest 24 matching landmarks are rejected as outliers under the assumption for an Affine transformation.

The thin plate spline transform (TPS) is used to estimate the nonlinear transformation between the fixed and moving image based on the robust landmark correspondence. The TPS transformation T is calculated by minimizing the binding energy E_{TPS}

$$E_{TPS} = \sum_i \|T(w_i) - v_i\|^2 + \lambda I_f \quad (5)$$

where

$$I_f = \int \int \left[\left(\frac{\partial^2 f}{\partial x^2} \right)^2 + 2 \left(\frac{\partial^2 f}{\partial x \partial y} \right)^2 + \left(\frac{\partial^2 f}{\partial y^2} \right)^2 \right] dx dy \quad (6)$$

with w_i and v_i denote the landmarks on the fixed and moving image, respectively.

TPS can provide a smooth matching function for each point in both images. The resulting nonlinear transformation is applied to map the moving image to the fixed image. For more details, we refer the readers to [11].

2.4 Image Registration Performance Bottleneck

The adaptive multi-resolution landmark based image registration algorithm can provide good registration results, but requires relatively time consuming point matching procedures. In Figure 3 we show the execution time profile for each step in our algorithm for a typical 2D (192×192) image pair registration. It is quite obvious that the bottleneck is the point matching step. However, as we mentioned before, the point matching procedure in our proposed algorithm

has a big advantage for easy parallelization by its design: *data independence*. Each landmark in the fixed image is independent to all the other landmarks, and its best match in the moving image is restricted to a certain area in the moving image. By fully utilizing this property, we propose to apply K -means data partitioning approach, and it has been successfully implemented on the IBM Cell Broadband Engine processor.

3 Parallelization on the Multicore Platform

The IBM Cell/B.E. [13,14] is a multicore chip with a relatively high number of cores. It contains a Power Processing Element (PPE) which has the similar function and configuration as the regular CPU. It also has multiple cores which are optimized for single precision float point algorithm, the Synergistic Processing Element (SPE). The PPE contains 32K L_1 cache, 512K L_2 cache and a large amount of physical memory (2G in our case). Unlike the PPE, the SPE has a quite different architecture compared with the standard CPU. The SPE operates on a 256KB local store to hold both code and data. The SPE also support 128 bit Single Instruction, Multiple Data (SIMD) instruction set for effective vector operations. The data transfer between SPE and PPE is through the direct memory access (DMA). DMA is quite time consuming so that a good parallel implementation should minimize the number of DMA operations.

3.1 Data Partitioning Using K-Means Clustering and Parallelization

Given all the detected landmarks in the fixed image, we first apply the K -means algorithm to cluster them based on their Euclidean distance in the image, where $K = 16$ is set to be the number of the computing units in the IBM Cell Blade machine. Based on the boundary landmarks in each cluster center, we can calculate the largest and smallest coordinates to crop the sub-image accordingly. Because we know the size of the code running on the SPE unit in advance, we can compute the maximal size of the sub-image that can be stored on a single core (e.g. single SPE). If the image patch can fit into the local storage, the whole cluster of landmarks and their corresponding image patch are sent to the SPE for parallel processing using just one direct memory access (DMA). Because the number of DMA is critical for the performance of the parallel algorithm, the advantage of applying K -means to group the landmarks into clusters is to minimize the number of direct memory access operations. Landmarks which are spatially close to each other are grouped together and transferred into one computing core (e.g. one SPE in a Cell Processor) using one DMA call. In Figure 4a we show the procedure of parallelizing the registration algorithm. The K -means clustering and job scheduling to 16 PPEs are illustrated in Figure 4b.

Thread for each core or Synergistic Processing Element (SPE) is created only once in order to reduce the overhead related to thread creation. The point matching for our registration implementation is executed twice, one during the stage of

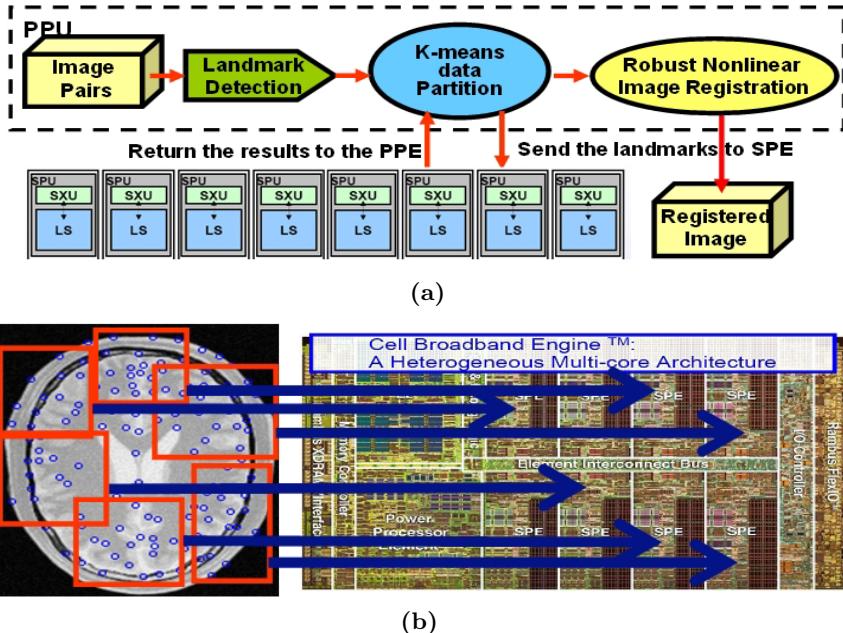


Fig. 4. The procedure of the parallel image registration algorithm. (a) The flow chart of the parallelization. (b) The data partitioning using K-means clustering.

linear registration and on during the stage of nonrigid registration. If we create an SPE thread each time when a point matching process starts, then all the SPE threads have to be created and destroyed repeatedly resulting additional execution time. In our implementation, all SPE threads are created before the first execution of the point matching process. These threads will be kept alive until the second point matching process is completed. It is true that the created SPE threads will be idle between the end of the first pass and the start of the second pass point matching processes, since the tasks in between are executed on PPE. This however does not cause any performance problem because those idle threads remain in the SPE spaces, which essentially has no impact to the main PPE thread.

Both the main core Power Processing Element (PPE) and SPEs are kept busy in sharing the point matching tasks. Initially we partitioned the landmark points into clusters and distribute evenly the clusters to the available SPEs. The main PPE thread waits for the completion of all active SPE threads before it proceeds to the next step of the registration process. Apparently this design overlooked the PPE computing resource, though it is indeed a different type of processor from SPE processors. It turns out that for some images, there may exist situations in which the sub-image enclosing the cluster is a little bit too large to send to a SPE by one DMA command and the space has to be further partitioned and then transferred to the SPE in more than one step. Even though such cases are rare, the second round partition is still critical to speed up the whole registration process.

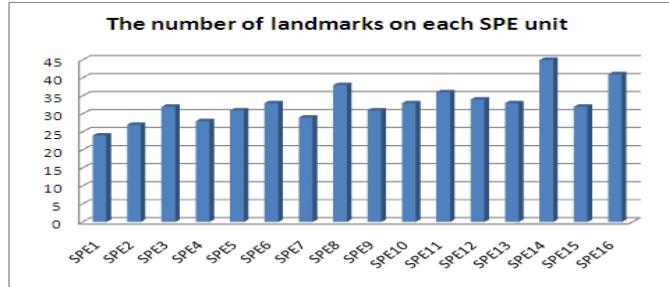


Fig. 5. The landmark distribution on each SPE unit of the IBM Cell/B.E.

The purpose of applying K -means clustering for data partitioning is to decrease the number of direct memory access (DMA) operations. In order to fully utilize each SPE computing unit, the work load should be balanced. Because our algorithm selects the landmarks considering their spatial relationships, the K -means clustering intends to provide similar amount of landmarks in each cluster. In Figure 5 we show a typical work load distribution for one image pair on 16 SPE, it is clear that the number of landmarks assigned to each cluster roughly form a uniform distribution.

3.2 Landmarks Assembling and Transformation Estimation

The main processor or main core (e.g. PPE) is responsible to spool and destroy all the threads of computing cores (e.g. SPE). As we shown in Figure 4a, the PPE is also in charge of assembling all the matching points returned from each SPE and converts the results back to the original image coordinate systems. Robust estimation is applied to reject outliers and preserve the robust landmark correspondence. Nonlinear transformation is finally estimated to register the fixed image and the moving image.

4 Experimental Results

The testing data used in our experiments were prepared in the department of Radiology, University of Medicine and Dentistry of New Jersey and ITK [15] public image repository. The dimensionality of the test image is 192×192 and the x and y resolution are 1.41 mm. We tested our algorithm using the simulated affine transformations. Forty simulated 2D human abdomen CT images are generated by applying forty simulated deformations. The algorithm is compared with the multiple resolution affine registration implemented in ITK [15] and also the free software MedINRIA developed by INRIA [16]. The registration accuracy is evaluated based on whether the algorithm can successfully recover the affine transformation parameters.

$$E = \max \left\{ \frac{|p_t^* - p_t|}{\delta_t}, \frac{|p_r^* - p_r|}{\delta_r}, \frac{|p_s^* - p_s|}{\delta_s} \right\}. \quad (7)$$

The p_t^*, p_r^*, p_s^* represent the estimated translation, rotation and scale parameters. The p_t, p_r, p_s are the ground true transformation parameters. The $\delta_t = 1, \delta_r = 0.5, \delta_s = 0.01$ are the normalization factors for translation, rotation and scale, respectively. The registration is treated as success if $E \leq 1.0$. Our proposed algorithm can recover 95% of the image pairs while the ITK and MedINRIA can only recover 70% and 50% for the image pairs with large deformation. From the experimental results we show that the proposed algorithm can accurately register two images even with 2.5 times scale difference and 45 degree of rotation. It is clear from this study that our algorithm demonstrate more robust registration for large deformations compared with commonly used registration algorithm [15][16]. Some representative results are shown in Figure 6.

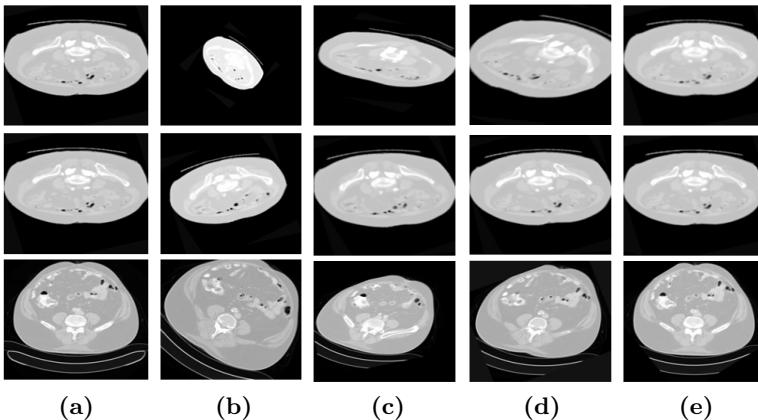


Fig. 6. Comparative registration results on the human abdomen CT image pair. (a) The fixed image. (b) The moving image. (c) The registration algorithm using the MedINRIA [16]. (d) The registration algorithm using ITK [15]. (e) The registration results using our algorithm.

The parallelization code was built for two different platforms. The parallel version was tested on an IBM BladeCenter Q21 featuring 2GB of RAM and two processors running at 3.2 GHz configured as a two-way, symmetric multiprocessor (SMP) [17]. A thread running on a PPE can communicate with all 16 SPEs. The Cell/B.E. SDK 3.0 and GCC compiler were used to implement and compile the algorithm. In all our experiments, there was one main thread running on one of the PPEs and up to 16 threads on the SPEs. The sequential version was running on an x86 machine at 2.6 GHz and 4G memory. The compiler is also GCC.

All the image pairs used for testing have dimensionality (192×192) . Because the point matching procedure dominates the running time of the registration algorithm, this step is the only part parallelized on the Cell/B.E. For fair comparison we run each implementation (sequential and parallel) 10 times. The statistics of the running speed on two platforms are shown in Table 1. Please notice that x86 refers to the sequential implementation on a x86 machine running with Linux.

Table 1. The statistics of the running speed for 10 trials using the sequential and parallel multicore platform

	Mean	Variance	Median
x86	5.95	0.26	5.82
Cell/B.E. (PPE only)	6.13	0.001	6.12
Cell/B.E. (16 SPEs)	0.60	0.0067	0.60
	Min	Max	80%
x86	5.33	6.93	6.22
Cell/B.E. (PPE only)	6.12	6.14	6.13
Cell/B.E. (16 SPEs)	0.50	0.70	0.70

The Cell/B.E. (PPE only) denotes the running time on the multicore Cell processor using only the main processor PPE. The Cell/B.E. (PPE only) represents the parallel running time by fully utilizing all the 16 computing cores (SPEs). The 80% column in Table 1 represents the sorted 80% smallest running times of all 10 trials, and is commonly used to evaluate the performance of the system. Using the multicore platform, we roughly achieved 10 times of speedup over its corresponding sequential implementation. In total, the parallel version of the algorithm can register a pair of image (192×192) in less than five seconds.

5 Related Work

While research effort continues to explore methods and strategies for more efficient and rapidly converging computational methods, increasing attention has been given to hardware architecture-based parallelization and optimization algorithms for the emerging high performance architectures, such as cluster and grid computing, advanced graphical processing units (GPU) and multicore computers. In [18] a parallel implementation of multimodal registration is reported for image guided neurosurgery using distributed and grid computing. The registration time was improved dramatically to near real-time. In [19] a distributed computation framework is developed, which allowed the parallelization of registration metrics and image segmentation/visualization algorithms while maximizing the performance and portability. One key deterring factor in adopting supercomputer-based, cluster-based or grid computing architectures is availability and cost. Even for some large clinical institutions, financial limitations can be a major hurdle. The recent emergence of low cost, high computing power, multi-core processor systems have opened up an alternative venue for developing cost-effective high performance medical image registration techniques. In [20], a close to real time implementation of a mutual information based linear registration is reported. The algorithm is designed based on the Cell Broadband Engine (Cell/B.E.) multicore processor architecture. General parallel data mining algorithms on the Cell/B.E. are reported in [21][22]. In [23], a high performance distributed sort algorithm is proposed for the Cell processor. As an

application, the implementation of the Radioastronomy image synthesis on the Cell/B.E. is discussed in [24]. According to our knowledge, this is the first study reporting fast and robust *landmark based nonlinear* image registration algorithm on a multicore platform.

6 Conclusion

In this paper, we have described a parallelization of a robust and accurate 2D image registration algorithm. The method is implemented on the IBM Cell/B.E. We have achieved about 10 times speed up, which allows our algorithm to complete the nonlinear registration of a pair of images (192×192) in less than five seconds. Our proposed data partitioning approach and the parallelization schema are independent to the parallel platforms and generic by its design, therefore it can be extended to other point matching related applications and other parallel platforms. The work discussed here is a part of an ongoing effort in developing full scale parallelization of a set of registration algorithms including the one reported in this paper. Future work will also include the generalization of the parallelization algorithm to handle 3D images.

Acknowledgement

This research was primarily conducted in IBM Watson Research Center funded by its internship program. It is also supported in part, by grants from the NIH through contract 5R01EB003587-04 from the National Institute of Biomedical Imaging and Bioengineering and contract 5R01LM009239-02 from the National Library of Medicine. Additional support was provided by IBM through a Shared University Research Award.

References

1. Maintz, J.B.A., Viergever, M.A.: A survey of medical image registration. *Medical Image Analysis* 2(1), 1–36 (1998)
2. Hill, D.L.G., Batchelor, P.G., Holden, M., Hawkes, D.J.: Medical image registration. *Physical Medical Biology* 46, 1–45 (1998)
3. Lester, H., Arridge, S.R.: A survey of hierarchical non-linear medical image registration. *Pattern Recognition* 32(1), 129–149 (1999)
4. Zitova, B., Flusser, J.: Image registration methods: A survey. *Image Vision Computing* 21(11), 977–1000 (2003)
5. Cena, B., Fox, N., Rees, J.: Fluid deformation of serial structural MRI for low-grade glioma growth analysis. In: Barillot, C., Haynor, D.R., Hellier, P. (eds.) *MICCAI 2004*. LNCS, vol. 3217, pp. 1055–1056. Springer, Heidelberg (2004)
6. Agostino, E., Maes, F., Vandermeulen, D., Suetens, P.: A viscous fluid model for multimodal non-rigid image registration using mutual information. *Medical Image Analysis* 7(4), 565–575 (2003)
7. Thirion, J.P.: Image matching as a diffusion process: An analogy with Maxwell demons. *Medical Image Analysis* 2(3), 243–260 (1998)

8. Vercauteren, T., Pennec, X., Perchant, A., Ayache, N.: Non-parametric diffeomorphic image registration with the demons algorithm. In: Ayache, N., Ourselin, S., Maeder, A. (eds.) MICCAI 2007, Part II. LNCS, vol. 4792, pp. 319–326. Springer, Heidelberg (2007)
9. Szeliski, R., Szeliski, R., Coughlan, J., Coughlan, J.: Hierarchical spline-based image registration. International Journal of Computer Vision, 194–201 (1994)
10. Fischler, M.A., Bolles, R.C.: Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. Comm. of the ACM 24, 381–395 (1981)
11. Chui, H., Rangarajan, A.: A new point matching algorithm for non-rigid registration. Computer Vision and Image Understanding 89(2), 114–141 (2003)
12. Levi, K., Weiss, Y.: Learning object detection from a small number of examples: the importance of good features. In: Proc. IEEE International Conference on Computer Vision and Pattern Recognition, Washington, DC, vol. 2, pp. 53–60 (2004)
13. Kahle, J.A., Day, M.N., Hofstee, H.P., Johns, C.R., Maeurer, T.R., Shippy, D.: Introduction to the cell multiprocessor. IBM J. Res. Develop. 49(4), 589–604 (2005)
14. Chen, T., Raghavan, R., Dale, J.N., Iwata, E.: Cell broadband engine architecture and its first implementation - a performance review. IBM J. Res. Develop. 51(5), 559–572 (2007)
15. ITK Software Guide: <http://www.itk.org>
16. MedINRIA: <http://www-sop.inria.fr/asclepios/software/medinria/>
17. Nanda, A.K., Moulic, J.R., Hanson, R.E., Goldrian, G., Day, M.N., D'Amora, B.D., Kesavarapu, S.: Cell/B.E. blades: Building blocks for scalable, read-time, interactive and digital media servers. IBM J. Res. Develop. 51(5), 573–582 (2007)
18. Chrisochoides, N., Fedorov, A., Kot, A., Archip, N., Black, P., Clatz, O., Golby, A., Kikinis, R., Warfield, S.: Toward real-time, image guided neurosurgery using distributed and grid computing. In: ACM/IEEE Super Computing (2006)
19. Warfield, S.K., Jolesz, F., Kikinis, R.: A high performance approach to the registration of medical imaging data. Parallel Computing 24(9), 1345–1368 (1998)
20. Ohara, M., Yeo, H., Savino, F., Gong, L., Inoue, H., Sheinin, V., Daijavad, S., Erickson, B.: Realtime murual information based linear registration on the cell broadband engine processor. In: Proc. International Symposium on Biomedical Imaging (2007)
21. Buehrer, G., Parthasarathy, S., Goyder, M.: Data mining on the cell broadband engine. In: International Conference on Supercomputing, pp. 26–35 (2008)
22. Duan, R., Strej, A.: Data mining algorithms on the cell broadband engine. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 665–675. Springer, Heidelberg (2008)
23. Gedik, B., Bordawekar, R.R., Yu, P.S.: Cellsort: high performance sorting on the cell processor. In: The 33rd international conference on very large databases, pp. 1286–1297 (2007)
24. Varbanescu, A.L., van Amesfoort, A.S., Cornwell, T., Mattingly, A., Elmegreen, B.G., van Nieuwpoort, R.V., van Diepen, G., Sips, H.J.: Radioastronomy image synthesis on the cell/B.E. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 749–762. Springer, Heidelberg (2008)

High Performance Matrix Multiplication on Many Cores

Nan Yuan^{1,2}, Yongbin Zhou^{1,2}, Guangming Tan¹, Junchao Zhang¹,
and Dongrui Fan¹

¹ Key Laboratory of Computer System and Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, P. R. China

² Graduate University of Chinese Academy of Sciences, Beijing 100039, P.R. China
{yuannan,ybzhou,jczhang,fandr}@ict.ac.cn, tgm@ncic.ac.cn

Abstract. *Moore’s Law* suggests that the number of processing cores on a single chip increases exponentially. The future performance increases will be mainly extracted from thread-level parallelism exploited by multi/many-core processors (MCP). Therefore, it is necessary to find out how to build the MCP hardware and how to program the parallelism on such MCP. In this work, we intend to identify the key architecture mechanisms and software optimizations to guarantee high performance for multithreaded programs. To illustrate this, we customize a dense matrix multiplication algorithm on Godson-T MCP as a case study to demonstrate the efficient synergy and interaction between hardware and software. Experiments conducted on the cycle-accurate simulator show that the optimized matrix multiplication could obtain 97.1% (124.3GFLOPS) of the peak performance of Godson-T.

1 Introduction

Previous techniques of increasing single thread performance through increased clock frequency and smarter architecture are now hitting the so-called *Power Wall* and *ILP Wall* [2]. Computer industry has widely consensus that future performance increases must largely come from increasing the number of processing cores on a die. This has led to swift changes in computer architectures in recent years: multi-core processors become commodity, and many-core processors [3][6][8][11] have entered the lexicon. Moore’s Law suggests that the number of on-chip processing cores doubles every generation. It is anticipated that future microprocessors will accommodate tens, hundreds or even thousands of processing cores, implicating that exploiting *thread level parallelism* (TLP) will become increasingly important.

To present what performance gains might be possible for multithreaded programs on MCPs, we choose to evaluate matrix multiplication on our proposed Godson-T MCP prototype. The matrix multiplication (MM) is a key building block of scientific and engineering applications, while at the same time representing a regular operation which is easy to reason about. Although MM seems very simple, previous work on implementing and optimizing high-performance

parallel MM on MCPs seems not very promising as expected. For instance, the performance of MM on Cyclops-64 processor is 13.9GFLOPS, which is 43.4% of the peak performance[6]. A systolic-like MM algorithm on TRIPS obtains 5.10 FLOPS/Cycle, which is 31.9% of the peak performance[5]. MM on Intel 80-core Terascale processor is observed to run only 37% of the peak performance[8].

The inefficiency of parallel MM kernels on these MCPs motivates us to rethink the challenges we confront when we design and program MCPs. Although many-core processors (MCPs) provide tremendous computational capability, extracting computational power on such processors effectively is not trivial as the feature sizes shrink. To our knowledge, the challenges at least include: (1) as the population of processing cores increase, on-chip data communication (especially the global communication) on the expanding interconnect will become more and more expensive. (2) Insufficient off-chip memory bandwidth makes the computational capability and memory accessing capability potentially unbalanced. The number of transistors per die is increasing at a faster rate than chip signal pins. This disparity will potentially limit the scalability of MCPs.

To address these problems, our method is not to reduce the latencies, but to tolerate them. We demonstrate a solution on *how to build a many-core processor with effective architectural supports*, and *how to develop a parallel algorithm benefits from the hardware*. Our solution presents an efficient synergy and interaction of software and hardware, resulting in nearly peak performance of matrix multiplication on Godson-T.

The remainder of the paper is organized as follows. Section 2 gives a brief introduction of our proposed Godson-T MCP, which we used for our evaluation. Section 3 illustrates the problem and our experimental methodology. Section 4 presents the key optimizations of matrix multiplication kernel, highlighting the underlying architectural supports. Section 5 concludes the paper.

2 Overview of Godson-T Many-Core Processor

Godson-T is a processor prototype of many-core architecture designed with 65nm CMOS technology. The early version of Godson-T has been introduced and evaluated in[10][13]. In this work, we make several improvements to support multithreaded program more efficiently. The overview of Godson-T is shown in Fig. 1, and more architecture details will be demonstrated in Sect. 4 when they are used.

As shown in Fig. 1(a), Godson-T has 64 homogeneous, dual-issue and in-order processing cores running at 1GHz. The 8-pipeline processing core supports 32-bit MIPS ISA (64-bit ISA will be supported in latter version) with synchronization instruction extensions. A floating-point multiply-accumulate operation can be issued to a fully-pipelined function unit in each cycle, so the peak floating-point performance of Godson-T is 128GFLOPS. Each processing core has a 16KB 2-way set-associative private instruction cache and a 32KB local memory. The local memory functions as a 32KB 4-way set-associative private data cache in default. It can also be configured as an explicitly-controlled and globally-addressed

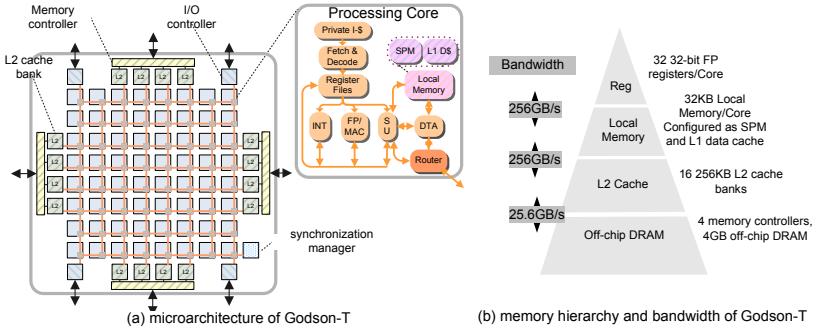


Fig. 1. An overview of Godson-T many-core processor

Scratched-Pad Memory (SPM), or a hybrid of cache and SPM. A DMA-like co-processor Data Transfer Agent (DTA) is built in each core for fast data communication. When the processing core is doing calculations, DTA can be programmed to manage various data communication patterns in background. In addition, there are 16 address-interleaved L2 cache banks (256KB each) distributed along the perimeter of the chip. The L2 cache is shared by all processing cores and can serve up to 64 outstanding cache accessing requests in total. Four L2 cache banks on the same side of the chip share a memory controller. The memory hierarchy of Godson-T and the data bandwidth between the neighboring levels of the memory hierarchy are shown in Fig. 1(b). A dedicated synchronization manager (SM) is a centralized unit to collect synchronization requests and handles them. It provides architectural support for fast mutual exclusion, barrier and single/wait synchronization. An 8 × 8 packet-switching 2-D mesh network connects all on-chip units. The 128-bit-width mesh network employs deterministic X-Y routing policy and provides total 2TB/s on-chip bandwidth among 64 processing cores.

3 Problem and Experimental Method

This work is an in-depth case study of optimizing SGEMM (Single-precision GEneral Matrix Multiplication) on 32-bit Godson-T. SGEMM is the BLAS 3 subroutines used to multiply two single-precision general dense matrices. It includes many variations (e.g. scaling), but we only consider a simple case: $C = A \times B + C$, where A , B and C are square matrices. Modern algorithms for SGEMM are block-based. To parallelize SGEMM, we decompose each of the three matrices into t^2 square blocks and assign each processing core the computation of a number of such blocks. The computation of a $C_{m,n}$ block requires t block multiplications and additions according to the following equation:

$$C_{m,n}+ = \sum_{k=0}^{t-1} A_{m,k} \times B_{k,n} \quad (1)$$

To exploit the spatial locality, it is best to assign the calculation of a $C_{m,n}$ to a processing core to minimize the number of blocks move around. The detailed algorithm is illustrated in section 4.

Experiments are conducted on Godson-T Architecture Simulator (GAS). GAS is an event-driven and cycle-accurate simulator for fast and precise simulation of Godson-T MCP. The major configurations are listed in Table 1.

Table 1. Major architectural parameters for experiments

Processing Core	64 cores running at 1GHz, dual-issue, load-to-use latency=3 cycles, and FMAC latency=4 cycles.
L1 I-Cache	16KB, 2-way set associative, 32B/cacheline, 1 cycle for hit.
Local Memory	Configured to 32KB SPM, 16 64-bit-width SRAM sub-banks with 2 memory ports each (1 for read, 1 for write), 1 cycle for load and store.
L2 Cache	16 banks, total 4MB, 8-way set-associative, 64B/cacheline, 4 cycles for contentionless and hit request.
Memory Controller & Off-chip DRAM	4 memory controllers, running at 1GHz. 64-bit FSB. Each memory controller controls 1GB DDR2-800 DRAM. DRAM clock is 400MHz, $t_{CAS} = 5$, $t_{RCD} = 5$, $t_{RP} = 5$, $t_{RAS} = 15$, $t_{RC} = 24$ measured in memory clock.
Mesh Network	2 cycles contentionless latency per hop.
Synchronization	Each synchronization request to SM and its acknowledgement from SM spend 6~66 cycles. Lock or Barrier request consumes another uncertain cycles until the synchronization dependence is resolved, e.g. barrier request should wait for barrier requests from all involved processing cores are collected in SM.

4 Customizing the Algorithm for Godson-T

In this section, we present the experience of optimizing a SGEMM algorithm on Godson-T. Subsection 4.1 focuses on the high-performance sequential blocked SGEMM kernel, which is groundwork of high-performance parallel SGEMM. Subsections 4.2 and 4.3 introduce our parallel algorithm of SGEMM, focusing on key architectural supports and software optimizations to address inefficient on-chip data communication and memory accessing. Subsection 4.4 discusses some related topics of our algorithm.

4.1 High-Performance Sequential SGEMM Kernel

In this subsection, we consider a sequential blocked SGEMM kernel, assuming that *all elements of matrix blocks are on the local memory when they are required*. To validate the premise, we configured the local memory of each processing core as a manually-controlled SPM, and load the required data onto SPM before using it. After that, loading and storing each element of the data on SPM consume

deterministic 1 cycle. Since each core has 32KB SPM, we selected the block dimension size 40×40 for A , B and C blocks in (1). The three blocked matrices occupy 18.75KB storage in total. The rest space of SPM is left for data communication between processing cores. We will explain it in the next subsection.

The sequential kernel for 40×40 SGEMM on one processing core is initially programmed in common 3 nested loops. Compiled by `gcc` version 4.3.2 with `-O3` optimization, the basic kernel performs only **0.16GFLOPS** even though all data resides in local SPM. A set of manual optimizations can be applied to the sequential kernel. With register tiling, we choose the tile size 2×4 , 4×2 , and 2×2 for A , B and C sub-blocks (notated as a , b and c), respectively. The innermost loop calculates the sub-block multiplication $c += a \times b$. Sub-blocks a , b and c consume 20 floating-point registers that can fit into 32 floating-pointer registers in MIPS ISA. Sub-block a is reused across the innermost loop. Leveraging on dual-issue pipeline of processing core, software pipeline and double buffering are applied to the innermost loop to eliminate pipeline stalls caused by register dependence. When the processing core calculates a sub-block multiplication, the core also stores the old result of c calculated in the last iteration, and load new b and c into spare 12 registers for the next sub-block multiplication. Since b and c use two sets of registers alternately, it is natural to unroll the innermost loop two times. Hence, the innermost loop calculates 2 sub-block multiplications. With carefully instruction scheduling, all memory access and other miscellaneous instructions (e.g. branch) can be issued and executed along with multiply-add instructions in the innermost loop. After these optimizations, the kernel performance significantly improves to **1.90GFLOPS**, which is **95%** of the peak performance. Since all effective calculations come from the innermost loop, it is better to enlarge the innermost loop times. We fuse the nested loops by encoding the strides of accessed sub-block addresses into 1-dimension arrays. Corresponding address stride of sub-blocks will be loaded out and added to current addresses to index next sub-blocks. After fusing two nested innermost loops into one, the innermost loop will iterate 100 times instead of 10 times. As a result, the kernel obtains nearly peak performance **1.99GFLOPS**. It could be found that this performance is about **12.4×** faster than the compiler-optimized one, which shows a lot of room of compilation technology.

4.2 Optimizing On-Chip Communication

Conventional inter-thread communication on shared memory is implicitly guaranteed by cache coherence protocols. However, communication through cache hierarchy is inefficient by reason of the indirect communication path and limited bandwidth of cache hierarchy. Besides **vertical** data communication in memory hierarchy (i.e. between different levels of memory hierarchy), Godson-T also supports **horizontal** data communication (i.e. between different SPMs) for efficient inter-thread communication, leveraging on low on-chip communication latency and tremendous on-chip bandwidth. This feature is naturally achieved by organizing all SPMs globally addressed and accessed, so that every core could directly access both local and remote SPMs with ordinary load/store

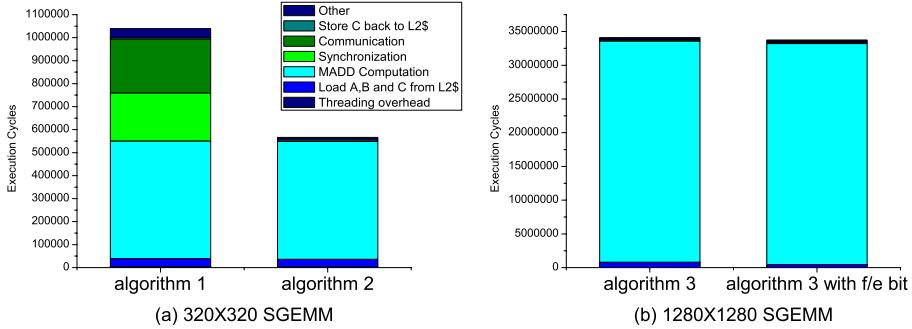


Fig. 2. Performance breakdown for different algorithms and input data sets. “Threading overhead” denotes the time for creating 64 threads. “Other” overhead includes the execution time of non-computational instructions, load imbalancing penalty, and etc.

instructions. Because of this, Cannon’s algorithm[4] can be easily implemented on the Godson-T’s 8×8 mesh network. In Cannon’s algorithm, most of threads only communicate with their neighboring nodes with horizontal communication, so that the network contention and the L2 cache accesses are minimized. Initially, matrices A , B and C are stored in normal row-major order arrays in off-chip memory. The algorithm for 320×320 SGEMM (where $t=8$) is described below.

Algorithm 1 - Calculating 320×320 SGEMM with 64 threads

-
- ① *thread_{i,j}* (ith row, jth column of the mesh) retrieves 40×40 matrix blocks $A_{i,(i+j)\%8}$, $B_{(i+j)\%8,j}$, $C_{i,j}$ from L2 cache into local SPM;
 - ② **for** (loop=1; loop \leq 8; loop++)
 - ③ *thread_{i,j}* calculates $C_{i,j} += A \times B$ on local SPM; // Sect. 4.1
 - ④ **if** (loop!=8) then // data communication for the next iteration
 - ⑤ barrier synchronization; // all threads complete computation
 - ⑥ *thread_{i,j}* gets A block from *thread_{(i+1)\%8,j}*’s SPM, and B block from *thread_{i,(j+1)\%8}*’s SPM to local SPM;
 - ⑦ *thread_{i,j}* stores $C_{i,j}$ from SPM back to L2 caches;
-

The simulation shows that **algorithm 1** obtains **63.06GFLOPS** on Godson-T, which is **49.27%** of the peak performance. The first column of Fig. 2(a) shows the performance breakdown.

It could be observed that communication and synchronization overhead cause 44.6% performance degradation. Note that SGEMM ought to be load-balanced theoretically. A large portion of synchronization overhead owes to imbalanced communication completion time: some threads are slower because of longer communication path. Therefore, inefficient communication is the major problem. This problem could be resolved on Godson-T by overlapping the cost of communication and computation. Data Transfer Agent (DTA) is built in each processing core to support asynchronous data communications. When the processing core is doing calculations, DTA can be programmed to manage data movements among

different memory parts in background. Compared to conventional DMA engine, DTA is a more advanced communication coprocessor for MCP. The first advantage is its versatility. DTA supports both horizontal and vertical data transfers. Moreover, DTA supports prefetch operation to command L2 caches to fetch bulk of data from off-chip memory into L2 caches. Besides accessing continuous data, DTA also supports 2-D stride data accessing. Hence, it is convenient to retrieve a blocked matrix from a large 2-D matrix array, while transposing it on-the-fly. The second advantage of DTA is its adaptation to network-on-chip. The delivered bandwidth of the network will degrade quickly if the injection bandwidth of the network is beyond some “saturation point” [9]. If DTAs send out their packets into network continuously and blindly, the on-chip network would probably be congested soon. DTA could assemble a group of requests or responses together into a single network packet for better utilization of bandwidth. Moreover, after sending a bunch of packets continuously, DTA sends a “probe” packet to the destination, and hangs until the “probe” packet is returned from the destination. Since the packet round-trip latency roughly reflects the contention status along the communication path, DTA can adaptively adjust the injection bandwidth to the network automatically with this simple flow-control mechanism.

Algorithm 2 - Calculating 320×320 SGEMM with 64 threads using DTA

```

① threadi,j retrieves  $40 \times 40$  matrix blocks  $A_{i,(i+j)\%8}, B_{(i+j)\%8,j}, C_{i,j}$  from
   shared L2 cache into local SPM using vertical DTA operations;
② for (loop=1; loop<=8; loop++)
③   barrier synchronization; // all data for communication is ready
④   if (loop!=8) then threadi,j gets  $A$  block from thread(i+1)\%8,j's SPM, and
       $B$  block from threadi,(j+1)\%8's SPM to local SPM using horizontal DTA
      operations; // in parallel with the next step
⑤   threadi,j calculates  $C_{i,j} += A \times B$  on local SPM (Sect. 4.1);
⑥ threadi,j stores  $C_{i,j}$  from SPM back to L2 caches;

```

The **algorithm 1** can be improved using DTA, described in **algorithm 2**. During the 1st step in **algorithm 2**, 64 threads retrieve data blocks from 16 L2 cache banks simultaneously, so that the network would be easily jammed. With network adaptation function of DTA, the utilized bandwidth of the network is **5.6**× larger than that of the naïve DTA without it. In the 4th step, we use non-blocked horizontal DTA operations to start the inter-thread communication, and overlap the overhead with local computation in the 5th step. The communication can be completely hidden in computation, so that the hard-wired global synchronization overhead in the 3rd step can be minimized to a few hundreds of cycles each time since the computation is load-balanced. The resultant performance dramatically improves to **115.70GFLOPS**, which is **90.39%** of the peak. The 2nd column of Fig. 2(a) shows the performance breakdown. Communication overhead is significantly reduced. Fetching data from off-chip memory becomes primary penalty and incurs 5.8% loss of the peak performance. It will be optimized in the next subsection.

4.3 Optimizing Memory Accessing

As stated previously, DTA is capable to command L2 caches to fetch data from the off-chip memory into L2 caches asynchronously. It helps improve performance by tolerating the off-chip memory latency. Larger SGEMM can be implemented based on 320×320 SGEMM in **algorithm 2**. For instance, the algorithm for $K \times K$ (K is a multiple of 320) SGEMM is shown in **algorithm 3**. Matrices A , B and C are initially stored in three row-major order arrays in the off-chip memory. There are two modifications compared to **algorithm 2**: (1) while all threads are calculating a 320×320 SGEMM during step 7, L2 caches are commanded by DTA prefetch operations in step 6 to fetch the data from off-chip memory into *least-recent-used* L2 cachelines for the next 320×320 SGEMM. (2) Reuse C matrix block in 320×320 SGEMM as much as possible, so it is unnecessary to load and store C blocks (step 5 and 8) every time.

Algorithm 3 - Calculating $K \times K$ SGEMM with 64 threads

```

①  $M = K/320;$ 
② for ( $r=0$ ;  $r < M$ ;  $r++$ ) // row
③   for ( $c=0$ ;  $c < M$ ;  $c++$ ) // column
④     for ( $s=0$ ;  $s < M$ ;  $s++$ )
⑤        $thread_{i,j}$  retrieves  $40 \times 40$   $A_{r \times 8+i, s \times 8+(i+j)\%8}$ ,  $B_{s \times 8+(i+j)\%8, c \times 8+j}$ 
          blocks from L2 caches into local SPM using DTA;  $40 \times 40$  block
           $C_{r \times 8+i, c \times 8+j}$  is retrieved if  $s==0$ ;
⑥       DTAs command L2 caches to prefetch  $320 \times 320$  block  $A$ ,  $B$  for the
          next iteration;  $320 \times 320$   $C$  block is prefetched if  $s==M-1$ ;
⑦       Compute  $320 \times 320$  SGEMM(algorithm 2); // in parallel with step 6
⑧       if ( $s==M-1$ ) then  $thread_{i,j}$  stores current  $C_{r \times 8+i, c \times 8+j}$  back to L2
          caches using DTA.

```

We simulate 1280×1280 SGEMM in our experiment. The off-chip prefetching in L2 caches can be completely overlapped by computation cycles in step 7. Therefore, when a 320×320 SGEMM calculation is completed, data for the next 320×320 SGEMM calculation is supposed to have already loaded into L2 caches. The resultant performance of the algorithm is **123.01GFLOPS**, which is **96.10%** of the peak performance. The first column of Fig. 2(b) shows the performance breakdown.

In these algorithms, we use hard-wired barrier synchronization in our algorithm: a thread sleeps after issuing a barrier request to synchronization manager (SM). After collecting all barrier requests, SM sends out acknowledgements to wake up all involved processing cores. Besides this coarse-grained synchronization mechanism, Godson-T also provides fine-grained full/empty-bit synchronization mechanism on SPM as implemented in [1][2], enabling to further hide the overhead of communication into computation. With it, a processing core could continue its computation as soon as the required element of the processing

instruction arrives. DTA coprocessor supports synchronized vertical and horizontal communication operating on full/empty bits as well.

The fine-grained synchronization scheme is used to tolerate memory latency in the 5th step of **algorithm 3**. Instead of using ordinary DTA operation, we use synchronized DTA operations to retrieve A , B and C blocks from L2 caches. When the data element arrives to SPM, the corresponding full/empty bit is set. Local synchronized load instructions in the sequential kernel guarantee that the processing core is stalled when the accessed memory location is “empty” and resumed as soon as the memory location is set to “full”. With this optimization, the memory latency of fetching data from L2 caches can be partly overlapped with kernel calculation. The resultant performance for 1280×1280 SGEMM increases to **124.33GFLOPS**, which is **97.14%** of the peak performance. The 2nd column of Fig. 2(b) shows the performance breakdown.

4.4 Discussion

Two-Level Latency-Tolerance and Horizontal Communication. It is observed the performance is primarily gained from two-level latency-tolerance mechanism and horizontal communication enabled by Godson-T architecture. With the two-level latency-tolerant framework, the on-chip network latency and the off-chip memory latency are expected to be hidden by useful computation. In SGEMM case we evaluated, both types of the latencies could be completely tolerated by computation, resulting in very efficient performance. The horizontal communication takes advantage of tremendous on-chip data bandwidth and on-chip data temporal locality. With the horizontal communication, data of a thread could be imported from on-chip SPMs besides shared low-level caches. It is complementary to latency-tolerance framework since it minimizes the cache hierarchy accesses that needs to be tolerated. Therefore, fetching data from neighboring SPMs is encouraged.

For a problem, we could construct an algorithm in two steps to handle memory latency and communication latency, respectively. At first, since the memory-processor gap remains a problematic issue, divide-and-conquer approach is chosen to exploit the best data temporal locality for each level of shared memory hierarchy (i.e. multi-level tiling). Vertical DTA operations are used to overlap the cost of on-chip shared cache accesses, off-chip memory accesses and the computation. Based on it, we parallelize the algorithm with multithreading. Inter-thread communication is realized by horizontal DTA operations among SPMs as much as possible. Therefore, a message-passing like algorithm is defined on the top of the memory hierarchy. The resultant algorithm effectively combines shared memory programming with one-sided message passing programming, two very distinct multithreaded programming paradigms in literature.

Performance Model. To generalize our algorithm with different machine or program configurations, we establish a performance model to study our parallel algorithm quantitatively. Let P^2 denote the number of processing cores and assume processing cores are distributed on a square 2-D mesh network. Let U

denote the size per SPM and L denote the size of L2 caches. Let O denote the DTA horizontal bandwidth, V denote the average DTA vertical bandwidth between a processing core and a L2 cache, B denote the bandwidth between last level caches and off-chip memory. Let $N \times N$ denotes the dimension of square matrix blocks computed on SPM, thus $PN \times PN$ denotes the dimension of square matrix blocks calculated in whole processor. At last, let S denote the width of data elements measured in byte ($S = 4$ for single-precision floating-point data and $S = 8$ for double-precision floating-point data).

The sequential blocked MM kernel could calculate a multiply-and-accumulate operation per cycle perfectly when all required data have already loaded onto SPM. Here, we only count the cost of memory accessing, communication and computation, and omit other insignificant overhead like synchronization and creating threads. The parallel efficiency is said efficient if both horizontal communication and off-chip memory prefetching overhead could be hidden into computation cycles. For a $K \times K$ (K is a multiple of N) GEMM problem, a thread in the algorithm fetches and stores $N \times N C$ blocks $K^2/P^2 N^2$ times, loads $N \times N A$ and B blocks $K^3/P^3 N^3$ times, and calculates K^3/P^2 multiply-accumulates, so the efficient efficiency is shown in (2). From it, it can be observed that it is critical to preserve delivered on-chip bandwidth V for better efficiency. For the configuration $P = 8$, $S = 4$, $N = 40$, $V \approx 1.5\text{B}/\text{cycle}$, $K = 1280$, the theoretical efficient efficiency is 97.96%. As K , P or N gets larger, the efficient efficiency can be better.

$$\text{Efficiency} = \frac{\frac{K^3}{P^2}}{\frac{2SN^2}{V} \times \frac{K^2}{P^2 N^2} + \frac{2SN^2}{V} \times \frac{K^3}{P^3 N^3} + \frac{K^3}{P^2}} = \frac{1}{\frac{2S}{V} \left(\frac{1}{K} + \frac{1}{PN} \right) + 1} \quad (2)$$

Since we apply double buffering and computation-communication overlapping at both SPM and L2 cache level, there are at least 4 constrained inequalities to ensure the efficient efficiency of the algorithm.

At SPM level, each SPM should be large enough to accommodate 5 matrix blocks for double buffering, so:

$$S \times 5N^2 \leq U \quad (3)$$

Inter-thread communication of A and B blocks should overlap with the computation of the sequential kernel in Sect. 4.1:

$$N^3 \geq S \times 2N^2/O \quad (4)$$

L2 caches also need to be large enough to accommodate C matrix blocks for current iteration and A , B and C matrix blocks for next iteration:

$$S \times 4P^2 N^2 \leq L \quad (5)$$

Prefetching A , B and C matrix blocks from off-chip memory to L2 caches should be overlapped with computation, so:

$$N^3 \times P \geq 3S \times N^2 P^2 / B \quad (6)$$

For the current configuration on Godson-T $P = 8, S = 4, N = 40, U = 32KB, L = 4MB, O = 16B/cycle$ and $B = 25.6B/cycle$, all inequalities are satisfied.

Extending for DGEMM. Calculating DGEMM requires more bandwidth and more storage space. For $S = 8$ and $U = 32KB$, N should not be larger than 28 to satisfy (3). Let $N = 28$, (3)~(6) are still satisfied under current Godson-T configuration. It means that our algorithm is also efficient for DGEMM. The efficiency for DGEMM is 94.69% for $K = 1260$ according to (2).

Scaling for More Cores. Inequalities (5) and (6) constrain the number of processing cores. To exploit the best utilization of SPM storage, let $S \times 5N^2 = U$ to maximize U in (3), then we get:

$$P^2 \leq 5L/4U, \text{ and } P^2 \leq B^2U/45S^3 \quad (7)$$

With current configuration of U, L and B , we could know that the number of processing cores P^2 is constrained to 160 by the size of L2 caches L for both SGEMM and DGEMM. However, it is anticipated that the size of L2 caches will be enlarged as well as the number of processing cores increases. Surprisingly, current off-chip bandwidth B shows a lot of room to support more cores. Our algorithm is not bandwidth bounded.

5 Conclusion

In this paper, we propose a new version of Godson-T processor of many-core architecture, and examined the adaptation of a high-performance matrix multiplication algorithm to Godson-T many-core processor. With architectural supports for latency-tolerance and horizontal communication on Godson-T, the overhead of on-chip communication and off-chip memory accessing are minimized. The cycle-accurate simulation shows that for 1280×1280 SGEMM, our algorithm achieves 124.3GFLOPS, which is about 97.1% of the peak performance. In theoretical analysis, we also show that our algorithm is also efficient for DGEMM and more processing cores.

This work proves that many-core processor is very promising to exploit thread-level parallelism efficiently. The approach is the combination of efficient architectural supports and corresponding programming methodology. We believe that many-core processors have great potential to serve as application accelerators, competitive to the existing accelerators [7][12] that are chiefly optimized for data-level parallelism. Our future work includes generalizing and automating the proposed two-level latency-tolerant programming methodology on Godson-T.

Acknowledgement. This work is partially supported by the National Grand Fundamental Research 973 Program of China under Grant No. 2005CB321600, the National Natural Science Foundation of China under Grant No. 60736012 and 60803030, EC under grant MULTICUBE FP7-216693, the Beijing Natural Science Foundation under Grant No.4092044, and the National High-Tech Research and Development Plan of China under Grant No.2009AA01Z103.

References

1. Alverson, R., Callahan, D., Cummings, D., Koblenz, B., Porterfield, A., Smith, B.: The Tera computer system. In: Proceedings of the 4th international conference on Supercomputing (1990)
2. Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., et al.: The landscape of parallel computing research: A view from berkeley. Electrical Engineering and Computer Sciences, University of California at Berkeley, Technical Report No. UCB/EECS-2006-183, December, 18(2006-183):19 (2006)
3. Burger, D., Keckler, S.W., McKinley, K.S., Dahlin, M., John, L.K., Lin, C., Moore, C.R., Burrill, J., McDonald, R.G., Yoder, W., et al.: Scaling to the End of Silicon with EDGE Architectures. Computer 37(7), 44–55 (2004)
4. Cannon, L.E.: A cellular computer to implement the Kalman filter algorithm (1969)
5. Diamond, J.R., Robatmili, B., Keckler, S.W., van de Geijn, R., Goto, K., Burger, D.: High performance dense linear algebra on a spatially distributed processor. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 63–72 (2008)
6. Hu, Z., del Cuivillo, J., Zhu, W., Gao, G.R.: Optimization of dense matrix multiplication on IBM cyclops-64: Challenges and experiences. In: Nagel, W.E., Walter, W.V., Lehner, W. (eds.) Euro-Par 2006. LNCS, vol. 4128, pp. 134–144. Springer, Heidelberg (2006)
7. Kapasi, U.J., Dally, W.J., Rixner, S., Owens, J.D., Khailany, B.: The Imagine stream processor. In: Proceedings 2002 IEEE International Conference on Computer Design, pp. 282–288 (2002)
8. Mattson, T.G., Van der Wijngaart, R., Frumkin, M.: Programming the Intel 80-core network-on-a-chip terascale processor. In: Proceedings of the 2008 ACM/IEEE conference on Supercomputing (2008)
9. Mukherjee, S.S., Silla, F., Bannon, P., Emer, J., Lang, S., Webb, D.: A comparative study of arbitration algorithms for the Alpha 21364 pipelined router. In: Proceedings of the 10th international conference on Architectural Support for Programming Languages and Operating Systems (2002)
10. Tan, G., Fan, D., Zhang, J., Russo, A., Gao, G.R.: Experience on optimizing irregular computation for memory hierarchy in manycore architecture. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 279–280 (2008)
11. Taylor, M.B., Kim, J., Miller, J., Wentzlaff, D., Ghodrat, F., Greenwald, B., Hoffman, H., Johnson, P., Lee, J.W., Lee, W., et al.: The Raw microprocessor: A computational fabric for software circuits and general-purpose programs. IEEE micro. 272, 2 (2002)
12. Williams, S., Shalf, J., Oliker, L., Kamil, S., Husbands, P., Yelick, K.: The potential of the cell processor for scientific computing. In: Proceedings of the 3rd conference on Computing Frontiers, pp. 9–20 (2006)
13. Ye, X., Nguyen, V.H., Lavenier, D., Fan, D.: Efficient parallelization of a protein sequence comparison algorithm on manycore architecture. In: Proceedings of the 9th international conference on Parallel and Distributed Computing, Applications and Technologies, pp. 167–170 (2008)
14. Zhu, W., Sreedhar, V.C., Hu, Z., Gao, G.R.: Synchronization state buffer: supporting efficient fine-grain synchronization on many-core architectures. In: Proceedings of the 34th annual International Symposium on Computer Architecture, pp. 35–45 (2007)

Parallel Lattice Basis Reduction Using a Multi-threaded Schnorr-Euchner LLL Algorithm

Werner Backes and Susanne Wetzel

Stevens Institute of Technology, Hoboken, NJ 07030, USA

Abstract. In this paper, we introduce a new parallel variant of the LLL lattice basis reduction algorithm. Our new, multi-threaded algorithm is the first to provide an efficient, parallel implementation of the Schorr-Euchner algorithm for today’s multi-processor, multi-core computer architectures. Experiments with sparse and dense lattice bases show a speed-up factor of about 1.8 for the 2-thread and about factor 3.2 for the 4-thread version of our new parallel lattice basis reduction algorithm in comparison to the traditional non-parallel algorithm.

1 Introduction

Lattice theory and in particular lattice basis reduction is of great importance in cryptography. Not only does it provide effective cryptanalysis tools but it is also believed to bring about new cryptographic primitives that exhibit strong security even in the presence of quantum computers [24][32]. In theory, many aspects of lattices are already well-understood. On the other hand many practical aspects, like the performance of lattice basis reduction algorithms, are still under investigation.

Lattice basis reduction algorithms try to find a *good* basis, i.e., a basis representing the lattice where the base vectors are not only as orthogonal as possible to each other, but also as short as possible. The LLL algorithm introduced by Lenstra, Lenstra and Lovász in [20] was the first algorithm to allow for an efficient computation of a fairly well-reduced lattice basis in theory but suffered from stability and performance issues in practice. In [33], Schnorr and Euchner introduced an efficient variant of the LLL algorithm, which could efficiently be used in practice, e.g., for cryptanalysis [33][29][30]. Since then, the main focus of research in lattice basis reduction was on improving the performance and stability of the reduction algorithms (e.g., [11][18][25][26]). While there is also some prior work on parallel lattice basis reduction, little to no further progress has been made in recent years—in particular in developing practical algorithms. A first line of parallelizing lattice basis reduction is mainly based on the original LLL algorithm [35][21][16][17][36][37]. Consequently, these solutions suffer from similar shortcomings in terms of stability and performance in practice as the original LLL algorithm. A second line of work is focused on vector computers [15][14][13][38], an architecture which is quite different from today’s mainstream compute servers.

In this paper, we present a new parallel LLL algorithm that overcomes the shortcomings of earlier work on parallel lattice basis reduction. Our new algorithm is based on

the Schnorr-Euchner algorithm and as such is the first—to the best of our knowledge—to provide an efficient parallel implementation for the Schnorr-Euchner algorithm. To date, it was believed to be impossible to efficiently parallelize the Schnorr-Euchner variant of the LLL algorithm due to the dependencies within the algorithm, which require frequent synchronization and data exchanges. In our new algorithm, this challenge is addressed by using a shared memory setting which allows us to replace time consuming inter-process communication with synchronization points (barriers) and locks (mutexes). In particular, using POSIX threads allows us to make effective use of today’s multi-processor, multi-core computer architecture. The high number of synchronization points in our new parallel algorithm forces us to take a new approach by means of concentrating on the balance of the workload in-between barriers among all threads in order to minimize the waiting time of each thread. It is important to note that this newly-developed approach also opens up new possibilities for parallelizing other algorithms which to date are believed to be difficult or impossible to parallelize.

Our implementation of the parallel LLL is optimized for reducing high-dimensional lattice bases with big entries that would require a multi-precision floating-point arithmetic to approximate the lattice basis if the original Schnorr-Euchner algorithm was used for the reduction. The reduction of these lattice bases is of great interest, e.g., for cryptanalyzing RSA [22][23][5][4]. In experiments with sparse and dense lattice bases, experiments with our new parallel LLL show (compared to the non-parallel algorithm) a speed-up factor of about 1.8 for the 2-thread and about factor 3.2 for the 4-thread version. The overhead of the parallel LLL decreases with increasing dimension of the lattice basis to less than 10% for the 2-thread and 4-thread version.

Outline. Section 2 provides the definitions and notations used in the remainder of the paper and introduces the Schnorr-Euchner algorithm. The main contributions of this paper are in Section 3. It describes our new algorithm, i.e., it shows in detail how the Schnorr-Euchner algorithm can be parallelized by using threads with locks and barriers. Section 4 introduces the sparse and dense lattice bases, discusses the parameters that control the parallel LLL reduction, and provides an analysis of our experiments. The paper closes with a discussion on directions for future work.

2 Preliminaries

A *lattice* $L \subset \mathbb{R}^n$ is an additive discrete subgroup of \mathbb{R}^n such that $L = \left\{ \sum_{i=1}^k x_i \underline{b}_i \mid x_i \in \mathbb{Z}, 1 \leq i \leq k \right\}$ with linear independent vectors $\underline{b}_1, \dots, \underline{b}_k \in \mathbb{R}^n$ ($k \leq n$). $B = (\underline{b}_1, \dots, \underline{b}_k) \in \mathbb{R}^{n \times k}$ is the *lattice basis* of L with dimension k . A lattice may have an infinite number of bases. Different bases B and B' for the same lattice L can be transformed into each other by means of a *unimodular transformation*, i.e., $B' = BU$ with $U \in \mathbb{Z}^{n \times k}$ and $|\det U| = 1$. Typical unimodular transformations are the exchange of two base vectors—referred to as swap—or the adding of an integral multiple of one base vector to another one—generally referred to as translation. The *determinant* $\det(L) = |\det(B^T B)|^{\frac{1}{2}}$ of a lattice is an invariant. The Hadamard inequality $\det(L) \leq \prod_{i=1}^k \|\underline{b}_i\|$ (where $\|\cdot\|$ denotes

the Euclidean length of a vector) gives an upper bound for the determinant of the lattice. Equality holds if B is an orthogonal basis. The *orthogonalization* $B^* = (\underline{b}_1^*, \dots, \underline{b}_k^*)$ of a lattice basis $B = (\underline{b}_1, \dots, \underline{b}_k) \in \mathbb{R}^{n \times k}$ can be computed by means of the Gram-Schmidt method: $\underline{b}_1^* = \underline{b}_1$, $\underline{b}_i^* = \underline{b}_i - \sum_{j=1}^{i-1} \mu_{i,j} \underline{b}_j^*$ for $2 \leq i \leq k$ where $\mu_{i,j} = \frac{\langle \underline{b}_i, \underline{b}_j^* \rangle}{\|\underline{b}_j^*\|}$ for $1 \leq j < i \leq k$ and $\langle \cdot, \cdot \rangle$ defines the scalar product of two vectors. It is important to note that for a lattice $L \subset \mathbb{R}^n$ with basis $B = (\underline{b}_1, \dots, \underline{b}_k) \in \mathbb{R}^{n \times k}$ a vector \underline{b}_i^* of the orthogonalization $B^* = (\underline{b}_1^*, \dots, \underline{b}_k^*) \in \mathbb{R}^{n \times k}$ is not necessarily in L . Furthermore, computing the orthogonalization B^* of a lattice basis using the Gram-Schmidt method strongly depends on the order of the basis vector of the lattice basis B . The *defect* of a lattice basis $B = (\underline{b}_1, \dots, \underline{b}_k) \in \mathbb{R}^{n \times k}$ defined as $\text{dft}(B) = \frac{\prod_{i=1}^n \|\underline{b}_i\|}{\det(L)}$ allows one to compare the quality of different bases. Generally, $\text{dft}(B) \geq 1$ and $\text{dft}(B) = 1$ for an orthogonal basis. The goal of lattice basis reduction is to determine a basis with as small a defect as possible. That is, for a lattice $L \subset \mathbb{R}^n$ with bases B and $B' \in \mathbb{R}^{n \times k}$, B' is better reduced than B if $\text{dft}(B') < \text{dft}(B)$. The most well-known and most-widely used lattice basis reduction method is the LLL reduction method [20]:

Definition 1. For a lattice $L \subseteq \mathbb{Z}^n$ with basis $B = (\underline{b}_1, \dots, \underline{b}_k) \in \mathbb{Z}^{n \times k}$, corresponding Gram-Schmidt orthogonalization $B^* = (\underline{b}_1^*, \dots, \underline{b}_k^*) \in \mathbb{Z}^{n \times k}$ and coefficients $\mu_{i,j}$ ($1 \leq j < i \leq k$), the basis B is LLL-reduced if (1) $|\mu_{i,j}| \leq \frac{1}{2}$ with $1 \leq j < i \leq k$ and (2) $\|\underline{b}_i^* + \mu_{i,i-1} \underline{b}_{i-1}^*\|^2 \geq y \|\underline{b}_{i-1}^*\|^2$ for $1 < i \leq k$.

The reduction parameter y may arbitrarily be chosen in $(\frac{1}{4}, 1)$. Condition (1) is generally referred to as size-reduction [731]. The Schnorr-Euchner algorithm [332] allows for an efficient computation of an LLL-reduced lattice basis in practice. By and large, Algorithm 1 is the original Schnorr-Euchner algorithm. In order to make LLL reduction practical, the Schnorr-Euchner algorithm uses floating-point approximations of vectors and the basis (APPROX_BASIS and APPROX_VECTOR) in Lines (1) and (22). For stability reasons, this in turn requires that the Schnorr-Euchner algorithm includes suitable measures in the form of correction steps (see [33] for details). These corrections include either the computation of exact scalar products (Line (6)) as part of the Gram-Schmidt orthogonalization or a step-back (Line (26)) due to a large μ_{ij} used as part of the size-reduction (Line (17)). In order to prevent the corruption of the lattice, the Schnorr-Euchner algorithm uses an exact data type to modify the actual lattice basis (Line (16)). (In Algorithm 1, p denotes the bit precision of the data type used to approximate the lattice basis.)

A modification for computing the Gram-Schmidt coefficients was introduced in [26] which was shown to allow for an increased accuracy of the orthogonalization. As part of our work, we adapted these modifications to the Schnorr-Euchner algorithm (Algorithm 1, Lines (3) - (12)). A second modification (originally introduced in [26]) is the checking of Condition (2) (Definition 1) followed by possibly swapping the respective lattice basis vectors (Algorithm 1, Lines (31) - (41)). We introduced both modifications to increase the overall performance and stability of the original Schnorr-Euchner algorithm.

Algorithm 1

INPUT: Lattice basis $B = (\underline{b}_1, \dots, \underline{b}_k) \in \mathbb{Z}^{n \times k}$
 OUTPUT: LLL-reduced lattice basis B

```

(1) APPROX_BASIS( $B'$ ,  $B$ )
(2) while ( $i \leq k$ ) do
(3)    $\mu_{ii} = 1$ ,  $R_{ii} = \|\underline{b}'_i\|$ ,  $S_i = R_{ii}$ 
(4)   for ( $1 \leq j < i$ ) do
(5)     if ( $|\langle \underline{b}'_i, \underline{b}'_j \rangle| < 2^{\frac{p}{2}} \|\underline{b}'_i\| \|\underline{b}'_j\|$ ) then
(6)        $R_{ij} = \text{APPROX\_VALUE}(\langle \underline{b}'_i, \underline{b}'_j \rangle)$ 
(7)     else
(8)        $R_{ij} = \langle \underline{b}'_i, \underline{b}'_j \rangle$ 

(19)    for ( $1 \leq m \leq j$ ) do
(20)       $\mu_{im} = \mu_{im} - \lceil \mu_{ij} \rfloor \mu_{jm}$ 
(21)      if ( $F_r = \text{true}$ ) then
(22)        APPROX_VECTOR( $\underline{b}'_i, \underline{b}_i$ )
(23)      if ( $F_c = \text{false} \wedge F_r = \text{true}$ ) then
(24)        RECOMPUTE_Rij()1
(25)       $F_r = \text{false}$ 
(26)      if ( $F_c = \text{true}$ ) then
(27)         $i = \max(i-1, 2)$ 
(28)       $F_c = \text{false}$ 
(29)    else
(30)       $i' = i$ 

(9)       $R_{ij} = R_{ij} - \sum_{m=1}^{j-1} R_{im} \mu_{jm}$ 
(10)      $\mu_{ij} = \frac{R_{ij}}{R_{jj}}$ 
(11)      $R_{jj} = R_{ii} - R_{ij} \mu_{ij}$ 
(12)      $S_{j+1} = R_{ii}$ 
(13)   for ( $i > j \geq 1$ ) do
(14)     if ( $|\mu_{ij}| > \frac{1}{2}$ ) then
(15)        $F_r = \text{true}$ 
(16)        $b_i = b_i - \lceil \mu_{ij} \rfloor b_j$ 
(17)     if ( $|\mu_{ij}| > 2^{\frac{p}{2}}$ ) then
(18)        $F_c = \text{true}$ 

(31)   while ( $((i > 1) \wedge (yR_{i-1, i-1} > S_{i-1}))$  do
(32)      $b_i \leftarrow b_{i-1}$ 
(33)     SWAP( $\underline{b}_i, \underline{b}_{i-1}$ )
(34)     SWAP( $\underline{b}'_i, \underline{b}'_{i-1}$ )
(35)      $i = i - 1$ 
(36)     if ( $i \neq i'$ ) then
(37)       if ( $i = 1$ ) then
(38)          $R_{11} = \|\underline{b}'_1\|$ 
(39)        $i = 2$ 
(40)     else
(41)        $i = i + 1$ 

```

3 Parallel LLL Reduction Using POSIX Threads

On a single computer system, programs can be parallelized by using multiple processes or threads [634]. The dependencies within the Schnorr-Euchner algorithm make (POSIX) threads the optimal choice for the parallelization due to the amount of data that has to be shared in the course of the reduction process. We are using the two techniques *mutex*² and *barrier* for the synchronization of critical memory access in the threads. A mutex² is a mutual-exclusion interface that allows to ensure that only one thread at a time is accessing critical data by setting (i.e., locking) the mutex on entering a critical code section and releasing (i.e., unlocking) it upon completion. Barriers are used to ensure that a number of threads that cooperate on a parallel computation wait for each other at a specific point in the algorithm before any of them is allowed to continue with further computations. These constructs seem to allow to parallelize arbitrary code as long as enough barriers and mutexes are used. However, the excessive use of barriers and mutexes has a negative effect on the overall running time, which consequently forces us to minimize the use of these constructs. This directly implies the need to identify as many independent (concurrent) code sequences as possible in order to allow for an efficient parallelization.

Using barriers and mutexes allows us to implement a new parallel lattice basis reduction algorithm which is well-suited for reducing bases of high dimensions and with large entries. This new parallel LLL is based on the Schnorr-Euchner algorithm (see Algorithm 1). The performance and limitations of the Schnorr-Euchner algorithm are dependent on the data type used for the approximation of the lattice basis. In case of

¹ RECOMPUTE_R_{ij} in Line (24) performs the same operations as shown in Lines (3) - (12).

² The word mutex is derived from mutually exclusive.

machine-type doubles, the reduction algorithm performs fairly well [33] [37] [12] and the reduction time is dominated by the operations using a long integer arithmetic. However, a major drawback in using doubles is the limitation in terms of dimension and bit length of lattice basis entries due to the fixed size of the double data type. One can avoid these restrictions by using a less efficient multi-precision floating point arithmetic. The reduction of lattice bases in high dimensions and entries of high bit length (which requires the use of a multi-precision floating point arithmetic for the approximation) are of interest in various context, e.g., for certain attacks on RSA [22] [23] [54]. Using a multi-precision floating-point arithmetic to approximate the lattice basis changes the running time behavior of the Schnorr-Euchner reduction algorithm dramatically. In this case, the operations on the approximate data type, e.g., in the Gram-Schmidt orthogonalization, are a major contributor to the overall running time of the Schnorr-Euchner algorithm. Thus, the main challenge is to parallelize these computations despite existing dependencies. Yet, operations on the exact data type (long integer arithmetic) are vector operations that can be parallelized more easily [13] [16] [17] [21] [35].

In striving to parallelize the reduction process, the dependencies within the Schnorr-Euchner algorithm (Algorithm 1) force us to keep the main structure of the algorithm. That is, one iteration of the main loop (Lines (2) - (41)) will be performed at the time. Furthermore, the overall structure of the main loop remains intact. It is possible to identify three main parts in the main loop, i.e, the orthogonalization (Lines (3) - (12)), the size-reduction (Lines (13) - (22)), and the condition-check part (Lines (31) - (41)). The order of these parts and the computations in between cannot be modified, i.e., in every iteration of the main loop, the computation of the orthogonalization has to be completed before one can do to the size-reduction and finally the checking of second LLL condition (Definition 1) after the size-reduction is finished. On the other hand, we show that the computations within each part can be modified or rearranged and eventually can be parallelized efficiently. The main contribution in this paper is in developing methods that allow to perform the computations of each part in parallel. A major challenge of this approach is to find means that balance computations well among all threads for all parallel parts in order to minimize the waiting times at the barrier in between these parts. It is crucial to minimize the amount of computations that cannot be parallelized, because they limit the maximum speed-up factor of the parallel algorithm according to Amdahl's law [6].

In our implementation of the parallel LLL, we were able to parallelize the orthogonalization and the size-reduction part. We are computing the scalar products separately from the orthogonalization to achieve a better overall balance. Unfortunately, the condition-check part cannot be parallelized. However, this is not a major drawback due to the small amount of computations that have to be performed in that part. In the following, we are detailing the concepts behind the parallelization of each part including rearrangements and modifications of the non-parallel computations these parts are based on. Our idea to concentrate on each parallelizable part in order to keep the balance among all threads at all time opens up new possibilities for parallelizing algorithms that are believed to be difficult or impossible to parallelize efficiently, like the Schnorr-Euchner algorithm.

3.1 Scalar Product Part

The computation of scalar products and exact scalar products in the course of the orthogonalization (Algorithm 1, Lines (3) - (12)) does not depend on prior computations within an iteration of the main loop. We therefore can divide the loop in Lines (4) - (12) into two loops, one that first computes the scalar products (Lines (5) - (8)) and a second that afterwards computes the remainder of the orthogonalization (Lines (9) - (12)). Thus, the computation of scalar products can be parallelized separately from the remainder of the orthogonalization. The major challenge in balancing the computation of scalar products is the need for computing exact scalar products under certain conditions for stability reasons. This means that assigning each thread a distinct, equal-sized segment of the iterations of the loop for the computation of the scalar products does not guarantee an equally distributed computation. Since there is no simple and efficient way to find a suitable partitioning, we instead divide iterations of the loop for the computation of scalar products and exact scalar products into small, distinct segments of size s_{sp} . Every thread requests a new segment as soon as it finishes the computation of its previous segment. This way, computationally intensive segments do not negatively impact the overall balance. We are using a shared segment position counter sl that is protected by a mutex mechanism to assign every thread a distinct segment. For each thread t as part of our parallel LLL algorithm the computation of scalar products looks as follows.³

Scalar Product – Thread_t

(1) $s = start_t$, $e = end_t$	(7) $else$
(2) while ($s \leq i$) do	(8) $R_{ij} = \langle \underline{b}_i', \underline{b}_j' \rangle$
(3) $e = (e > i) ? i : e$	(9) MUTEX_LOCK (l_1)
(4) for ($s \leq j < e$) do	(10) $s = sl$
(5) if ($ \langle \underline{b}_i', \underline{b}_j' \rangle < 2^{\frac{p}{2}} \ \underline{b}_i'\ \ \underline{b}_j'\ $) then	(11) $sl = sl + s_{sp}$
(6) $R_{ij} = \text{APPROX_VALUE}(\langle \underline{b}_i, \underline{b}_j \rangle)$	(12) $e = sl$
	(13) MUTEX_UNLOCK (l_1)

The locking mechanism (**MUTEX_LOCK** and **MUTEX_UNLOCK**) is placed around the updates for the segment position counter, which is always increased by s_{sp} to ensure that no two threads are processing the same segment and therefore are accessing non-overlapping entries in the R -matrix. The initial segment for a thread is given by $start_t$ and end_t , and the starting value for sl is determined by the maximum of the end_t .

3.2 Orthogonalization Part

Given the parallel scalar product computation, Algorithm 3 shows the remaining part of the orthogonalization. To increase the visibility of the dependency issues in the computation of R_{ij} , we replaced the sum $\sum_{m=1}^{j-1} R_{im} \mu_{jm}$ of Line (9) in the Schnorr-Euchner algorithm with the appropriate loop that is used in practice. One can clearly see (Algorithm 3, Lines (2) - (3)) that all R_{im} (with $1 \leq m < j$) are needed before one can compute R_{ij} . It is important to note that all μ_{jm} (with $1 \leq m < j$) have already been

³ In outlining our multi-threaded programs, we distinguish between variables that are local for every thread and variables that are shared among all threads. Local variables are highlighted by the use of a different font (e.g., `local` vs. `shared`).

computed. As is, the orthogonalization in Algorithm 3 is hard to parallelize. But a parallelized computation of R_{ij} is the key to successfully balance the work among all threads. With Algorithm 4 we have developed a modification of Algorithm 3 that allows to take advantage of the already computed μ_{jm} . Algorithm 4 introduces an additional value r_j to help with the computation of R_{ij} . The final value of r_j is in fact $\sum_{m=1}^{j-1} R_{im}\mu_{jm}$ known from the Schnorr-Euchner algorithm, but the value is computed in a different, more parallel-friendly fashion. Intuitively speaking, instead of computing the sum horizontally like in the Schnorr-Euchner algorithm, Algorithm 4 uses a vertical approach and updates the sum in r_j in phases, which eventually allows for a parallelization of the computation of r_j .

Algorithm 3:

```

(1) for  $(1 \leq j < i)$  do
(2)   for  $(1 \leq m < j)$  do
(3)      $R_{ij} = R_{ij} - R_{im}\mu_{jm}$ 
(4)      $\mu_{ij} = \frac{R_{ij}}{R_{jj}}$ 
(5)      $R_{ij} = R_{ij} - R_{ij}\mu_{ij}$ 
(6)      $S_{j+1} = R_{ii}$ 

```

Algorithm 4:

```

(1) for  $(1 \leq j < i)$  do
(2)   for  $(j \leq l < i)$  do
(3)      $r_l = r_l + R_{lj-1}\mu_{lj-1}$ 
(4)      $R_{ij} = R_{ij} - r_j$ 
(5)      $\mu_{ij} = \frac{R_{ij}}{R_{jj}}$ 
(6)      $R_{ij} = R_{ij} - R_{ij}\mu_{ij}$ 
(7)      $S_{j+1} = R_{ii}$ 

```

In the following, we are now using Algorithm 4 as starting point for describing our parallelization of the orthogonalization. In particular, we detail our new methods on how to compute the r_j (Algorithm 4, Lines (2) - (3)) using multiple threads. A straight-forward approach to parallelize Algorithm 4 would be to put barriers around the computation of r_j and splitting up the iterations of the inner loop (Algorithm 4, Lines (2) - (3)) in between to be computed by all threads. The barriers allow one of the threads to compute R_{ij} , R_{ii} , μ_{ij} and S_{j+1} correctly, while the others wait for the next iteration of the outer loop (Algorithm 4, Lines (1) - (7)) in the orthogonalization. However, the excessive use of barriers is a major drawback which makes this parallelization attempt unusable in practice.

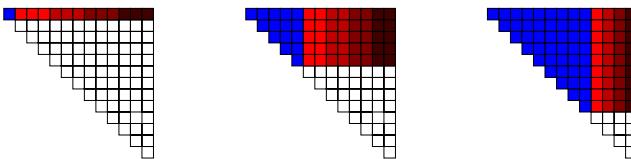


Fig. 1. Parallel (each shade of red represents one thread) and non-parallel (blue) computations for main and helper threads

Figure 1 shows the intuition for our new approach for the parallelization of the orthogonalization. A main (or control) thread computes the necessary non-parallelizable computations (blue) first, and then distributes the parallel computations (shades of red) among all threads. Obviously, the size of the parallel part depends on the size of the non-parallel part, and it is crucial to find an optimal balance between the two. Increasing the non-parallel part decreases the number of barriers. However, at a certain point this may also decrease the workload per thread.

Orthogonalization – Thread₁ (main)

```

(1) j = 0
(2) while (j < i) do
(3)   s = j, m = 0
(4)   while (m < so  $\wedge$  j < i) do
(5)     for (s  $\leq$  l < j) do
(6)        $r_j = r_j - R_{il}\mu_{jl}$ 
(7)        $R_{ij} = R_{ij} - r_j$ 
(8)        $\mu_{ij} = \frac{R_{ij}}{R_{jj}}$ 
(9)        $R_{ii} = R_{ii} - R_{ij}\mu_{ij}$ 
(10)       $S_{j+1} = R_{ii}$ 
(11)      m = m + 1, j = j + 1
(12)      COMPUTE_SPLIT_VALUES1(split)
(13)      BARRIER_WAIT(b1)

```

```

(14)      for (j  $\leq$  l < split1) do
(15)        for (s  $\leq$  m < j) do
(16)           $r_l = r_l - R_{lm}\mu_{lm}$ 

```

Orthogonalization – Thread_t

```

(1) e = 0
(2) while (e < i) do
(3)   s = e, e = e + so
(4)   if (e > i) then
(5)     e = i
(6)   BARRIER_WAIT(b1)
(7)   for (splitt  $\leq$  l < splitt+1) do
(8)     for (s  $\leq$  m < e) do
(9)        $r_l = r_l - R_{lm}\mu_{lm}$ 

```

In every iteration of the loop in Lines (4) - (11), the main thread computes only a small segment of size *s_o* of the orthogonalization. The size of this segment determines the number of barriers (function BARRIER_WAIT) and the amount of work that can be computed in parallel (Thread₁, Lines (14) - (16) and Thread_t, Lines (7) - (9)) in each iteration. Further details on the parameter choices and their effect on the computational balance are discussed in Section 4.2. Compared to the straight-forward parallelization idea, our new approach now uses one instead of two barriers for each iteration of the outer loops (Thread₁, Lines (2) - (16) and Thread_t, Lines (2) - (9)). Computing small segments of the orthogonalization significantly reduces the number of iterations of the outer loops and number of barriers needed for the orthogonalization. It is important to note that although not explicitly mentioned here in this paper (to improve readability), we have implemented measures to ensure that variables are not unintentionally overwritten by the subsequent loop iterations. The function COMPUTE_SPLIT_VALUES₁ computes the *split_t* values that are responsible for balancing the parallel computation of *r_l* among all threads. In addition, the splitting has to ensure that the main thread computes the necessary *r_l* for the next iteration of its outer loop. The splitting among all threads is not necessarily even.

3.3 Size-Reduction Part

In the non-parallel version of the Schnorr-Euchner algorithm, the size-reduction (Algorithm 5) also contains a part that updates the μ -coefficients. Due to the fact that the actual operations on the exact and approximate lattice basis are simple to parallelize, our parallelization computes the μ -update separately. Consequently, Algorithm 6 has two parts, the μ -update (Lines (1) - (8)) and lattice basis update (Lines (9) - (13)). Our solution furthermore introduces an additional array *f* to store the μ_{ij} values that are necessary for the update of the lattice basis. For the first part of Algorithm 6, we adopt a similar strategy to the one used for the orthogonalization (Section 3.2). In contrast to the orthogonalization, we do not need to modify the loop (Algorithm 6, Lines (7) - (8)) that updates the μ -coefficients in order to parallelize it. The dedicated main thread, like in the orthogonalization part, updates only small segments of size *s_μ* of the μ_{im} . In addition, it performs the computations that cannot be done in parallel (Algorithm 6,

Lines (2) - (6)) and distributes the computation among all threads for the parallel parts (Thread₁, Lines (20) - (24) and Thread_t, Lines (4) - (8)).

Algorithm 5:

```

(1) for ( $i > j \geq 1$ ) do
(2)   if ( $|\mu_{i,j}| > \frac{1}{2}$ ) then
(3)      $F_r = \text{true}$ 
(4)      $b_j = b_i - \lceil \mu_{i,j} \rceil b_j$ 
(5)     if ( $|\mu_{i,j}| > 2^{\frac{p}{2}}$ ) then
(6)        $F_c = \text{true}$ 
(7)       for ( $1 \leq m \leq j$ ) do
(8)          $\mu_{im} = \mu_{im} - \lceil \mu_{i,j} \rceil \mu_{jm}$ 
(9)       if ( $F_r = \text{true}$ ) then
(10)      APPROX_VECTOR( $\underline{b}'_i, \underline{b}'_j$ )

```

Algorithm 6:

```

(1) for ( $i > j \geq 1$ ) do
(2)    $f_j = \mu_{ij}$ 
(3)   if ( $|\mu_{ij}| > \frac{1}{2}$ ) then
(4)      $F_r = \text{true}$ 
(5)     if ( $|\mu_{ij}| > 2^{\frac{p}{2}}$ ) then
(6)        $F_c = \text{true}$ 
(7)       for ( $1 \leq m \leq j$ ) do
(8)          $\mu_{im} = \mu_{im} - \lceil \mu_{ij} \rceil \mu_{jm}$ 
(9)       for ( $i > j \geq 1$ ) do
(10)         if ( $|f_j| > \frac{1}{2}$ ) then
(11)            $b_i = b_i - \lceil f_j \rceil b_j$ 
(12)         if ( $F_r = \text{true}$ ) then
(13)           APPROX_VECTOR( $\underline{b}'_i, \underline{b}'_j$ )

```

μ -Update – Thread₁ (main)

```

(1)  $j = i - 1, j_e = 0$ 
(2) while ( $j \geq 1$ ) do
(3)    $j_s = j, m = 0$ 
(4)   while ( $m < s_\mu \wedge j \geq j_e$ ) do
(5)     for ( $j_s \geq 1 > j$ ) do
(6)        $\mu_{ij} = \mu_{ij} - \lceil f_{j_s} \rceil \mu_{j_s}$ 
(7)       if ( $|\mu_{ij}| > \frac{1}{2}$ ) then
(8)          $f_j = \mu_{ij}$ 
(9)          $F_r = \text{true}$ 
(10)        if ( $|\mu_{ij}| > 2^{\frac{p}{2}}$ ) then
(11)           $F_c = \text{true}$ 
(12)           $\mu_{ij} = \mu_{ij} - \lceil \mu_{ij} \rceil$ 
(13)           $m = m + 1$ 
(14)        else
(15)           $f_j = 0$ 
(16)         $j = j - 1$ 
(17)       $j_c = j$ 

```

μ -Update – Thread₂ (split)

```

(18) COMPUTE_SPLIT_VALUES2( $split$ )
(19) BARRIER_WAIT( $b_2$ )
(20)  $j_e = split_T$ 
(21) for ( $j_s \geq m > j_e$ ) do
(22)   if ( $f_m \neq 0$ ) then
(23)     for ( $split_T \leq 1 < split_{T+1}$ ) do
(24)        $\mu_{il} = \mu_{il} - \lceil f_m \rceil \mu_{ml}$ 

```

The difference in computations for the main and helper thread compared to the orthogonalization might require a different splitting. We therefore use the function COMPUTE_SPLIT_VALUES₂ to compute the split values. The check for $f_m \neq 0$ (Thread₁, Line (22) and Thread_t, Line (6)) ensures, that the μ coefficients are only updated if the corresponding $|\mu_{ij}| > \frac{1}{2}$. The second part of Algorithm 6 (Lines (9) - (13)), namely the actual size-reduction step or the update of the exact and approximate lattice basis, can easily be computed in parallel. We only need to split up every vector operation into equal sized parts. This way we can avoid dependencies that would require the use of barriers or mutexes. In contrast to the scalar product part (Section 3.1) we do not have to deal with a phenomenon like the unpredictable behavior of exact scalar products that would suggest splitting up the vector operations into smaller segments. For T being the number of threads and n the dimension of a lattice basis vector, the implementation for Thread_t of the parallel size-reduction for our parallel LLL looks as follows:

Size Reduction – Thread_t

```

(1) if ( $F_r = \text{true}$ ) then
(2)    $s = \lceil \frac{(t-1)n}{T} \rceil, e = \lceil \frac{tn}{T} \rceil$ 
(3)   for ( $i-1 \geq j \geq 1$ ) do
(4)     if ( $|f_j| > \frac{1}{2}$ ) then

```

```

(5)       for ( $s \leq 1 \leq e$ ) do
(6)          $b_{il} = b_{il} - \lceil f_j \rceil b_{jl}$ 
(7)       for ( $s \leq 1 \leq e$ ) do
(8)          $b'_{il} = \text{APPROX\_VALUE}(b_{il})$ 
(9)        $F_r = \text{false}$ 

```

4 Experiments

4.1 Lattice Bases

The experiments in this paper were performed using sparse and dense lattice bases in order to show that our parallel (i.e., multi-threaded) LLL reduction algorithm performs and scales well on different types of lattice bases. We have chosen knapsack lattices [19, 9, 8, 28, 10] and random lattices (as defined by Goldstein and Mayer [12, 27]) as examples for sparse lattice bases. Unimodular lattice bases serve as representatives for dense lattice bases. Previous experiments [37, 2] showed that unimodular lattice bases are more difficult to reduce than, e.g., knapsack lattices given the same dimension and maximum length of the basis entries.

For our experiments, we generate the unimodular lattice bases as $B_u = V \cdot U$ with a lower triangular matrix U and upper triangular matrix V . The entries in the diagonal of U and V are set to 1 while the lower (respectively upper) part of the matrix is selected uniformly at random. We generated 100 unimodular lattice bases with a maximum bit length $b = 1000$ for each dimension $n \in [50, 1000]$ in steps of 50. Knapsack lattices (see Figure 2) have been developed in the context of solving the subset sum problems [19, 9, 8, 28]. For our experiments we have defined $W = \lceil \sqrt{n} \rceil + 1$. The weights a_1, \dots, a_n have been chosen uniformly at random from $(1, 2^b]$. The sum S has been computed by adding up half of the weights that we have selected at random. For Goldstein-Mayer random lattice bases (see Figure 3) with prime p (of bit length b), x_1, \dots, x_n chosen independently and uniformly from $[0, p - 1]$. We generated 100 knapsack lattices and Goldstein-Mayer random lattices for each dimension $n \in [50, 1500]$ in steps of 50 and maximum bit length $b \in \{1000, 2500\}$.

$$B_k = \begin{pmatrix} 2 & \cdots & 0 & 1 \\ \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & 2 & 1 \\ a_1 W & \cdots & a_n W & SW \\ 0 & \cdots & 0 & -1 \\ W & \cdots & W & \frac{n}{2} W \end{pmatrix} \quad B_r = \begin{pmatrix} p & x_1 & \cdots & x_{n-1} & x_n \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \\ 0 & 0 & \cdots & 0 & 1 \end{pmatrix}.$$

Fig. 2. Knapsack lattice basis

Fig. 3. Goldstein-Mayer random lattice basis

4.2 Parameters

In this section, we discuss the parameters introduced in Section 3 that are responsible for balancing computations among all threads. As discussed before, every part of the multi-threaded LLL has to be balanced. The main loop in the multi-threaded LLL algorithm would otherwise multiply the waiting times and therefore every major imbalance would decrease the maximum speed-up factor.

The scalar product part introduces the parameters $start_t$ and end_t to minimize the amount of mutexes by assigning each thread a fixed first segment of scalar products to be computed. The most important parameter is the size s_{sp} of the subsequent segment. Bigger segments would decrease the number locks. Smaller segments, on the other hand, minimize the time needed for a single segment which leads to a more balanced computation. In the orthogonalization part we use s_o for the size of a segment in the

computation of R_{ij} . For smaller values of s_o the increased number of barriers would impact the overall running time significantly. We also observed an increase in the system time. Larger values, on the other hand, result in a bigger part being computed by the main thread only which has a negative impact on the speed-up factor. The split values computed by COMPUTE_SPLIT_VALUES₁ play an important role in the overall balancing of the computation. A slight imbalance in the partitioning could give the main thread the extra time to already perform the additional computations for the next iteration of the main loop. For the μ -update within the size-reduction part the situation is similar to the orthogonalization part. The size s_μ of a segment of μ_{ij} to be updated has the same effect as the size s_o has on the orthogonalization. Nevertheless, the optimal value for s_μ and the split values (COMPUTE_SPLIT_VALUES₂) may be different.

We have determined suitable parameter choices by performing small-scale tests with our new parallel LLL algorithm. In our experiments, we use $s_{sp} = s_o = s_\mu = 16$. For the split values (COMPUTE_SPLIT_VALUES₁ and COMPUTE_SPLIT_VALUES₂) we confirmed that a slight imbalance is indeed beneficial. For the 2-thread case, the split-up is 47.5% for the main and 52.5% for the other thread. In the 4-thread version, we modified the split-up to 22% for the main and 26% for each of the other threads.

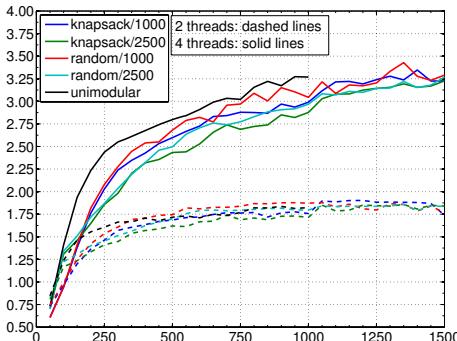
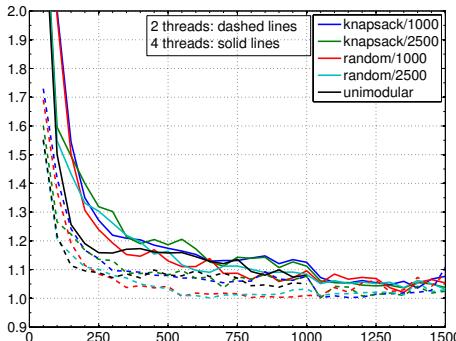
4.3 Results

The experiments in this paper were performed on Sun X2200 servers with two dual core AMD Opteron processors (2.2 GHz) and 4 GB of main memory using Sun Solaris. We compiled all programs with GCC 4.1.2 using the same optimization flags. For our implementation of the parallel and non-parallel LLL algorithm we used GMP 4.2.2 as long integer arithmetic and MPFR 2.3.1 as multi-precision floating-point arithmetic for the approximation of the lattice basis. The experiments were performed using MPFR with 128 bit of precision. Based on our setup we conducted experiments with the non-parallel Schnorr-Euchner algorithm (Algorithm 1) and the 2-thread and 4-thread version (Section 3) of our newly developed implementation of the parallel LLL algorithm.

We used `gettimeofday` to measure the actual time (real time) that the LLL algorithms needed for the reduction. The function `getrusage` is used to determine how much processor time (user time) and system time was necessary. For non-parallel (or single-threaded) applications, real time is, depending on the system load, roughly the sum of user and system time. But in case of multi-threaded applications, `getrusage` returns the sum of user and system time for all threads. We therefore use real time for the single and multi-threaded LLL algorithm to determine the scaling or speed-up factor for our new multi-threaded implementation. Comparing the sum of user and system time for the single and multi-threaded LLL allows us to compute the necessary overhead for the multi-threaded version.

Figure 4 shows the average speed-up factor per dimension for the new multi-threaded LLL compared to the non-parallel variant. The speed-up (i.e., scaling factor) is the quotient of the real time of the non-parallel version and the real time of the multi-threaded version. The speed-up factor increases with the dimension up to around 1.8 for the 2-thread and factor 3.2 for the 4-thread version⁴ of our new parallel LLL algorithm.

⁴ The running time drops from about 17.5h to about 5.5h for knapsack/2500 at dimension 1500.

**Fig. 4.** Speed-up for multi-threaded LLL**Fig. 5.** Overhead for multi-threaded LLL

It is interesting to note that for the selected parameters the speed-up factor for unimodular lattice bases increases faster than for knapsack or Goldstein-Mayer random lattices, and eventually all three speed-up factors reach a similar maximum value, as one can clearly observe in the 2-thread case. The 2-thread version, as expected, reaches its maximum earlier than the 4-thread version. Figure 5 shows the average overhead per dimension caused by the new parallel LLL algorithm compared to the non-parallel Schnorr-Euchner algorithm. We define the overhead as the quotient of the sum of user and system time of all threads to the sum of user and system time for the non-parallel version. The overhead of the new parallel LLL decreases with increasing dimension of the lattice basis to less than 10% for the 2-thread and the 4-thread version.

5 Conclusion and Future Work

In this paper we introduced a new parallel LLL reduction algorithm based on the Schnorr-Euchner algorithm. We used POSIX threads to parallelize the reduction in a shared memory setting. In our experiments we show that our new variant scales well and achieves a considerable decrease in the running time of the LLL reduction by taking advantage of today's computers multi-core, multi-processor capabilities.

Future work includes adjusting our algorithms to more than 4 threads and finding the corresponding heuristics for the selection of the parameters that control the balancing among all threads. In addition, we plan to explore the possibility to parallelize the LLL Gram using buffered transformations [3]. The structure and the required updates of the Gram matrix pose the main challenge in parallelizing this algorithm. We are also looking into the possibility of developing a parallel LLL for the GPU of graphics cards.

References

1. Backes, W., Wetzel, S.: New Results on Lattice Basis Reduction in Practice. In: Bosma, W. (ed.) ANTS 2000. LNCS, vol. 1838, pp. 135–152. Springer, Heidelberg (2000)
2. Backes, W., Wetzel, S.: Heuristics on Lattice Basis Reduction in Practice. ACM Journal on Experimental Algorithms 7 (2002)

3. Backes, W., Wetzel, S.: An Efficient LLL Gram Using Buffered Transformations. In: Ganzha, V.G., Mayr, E.W., Vorozhtsov, E.V. (eds.) CASC 2007. LNCS, vol. 4770, pp. 31–44. Springer, Heidelberg (2007)
4. Bleichenbacher, D., May, A.: New Attacks on RSA with Small Secret CRT-Exponents. In: Yung, M., Dodis, Y., Kiayias, A., Malkin, T.G. (eds.) PKC 2006. LNCS, vol. 3958, pp. 1–13. Springer, Heidelberg (2006)
5. Blömer, J., May, A.: A Tool Kit for Finding Small Roots of Bivariate Polynomials over the Integers. In: Cramer, R. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 251–267. Springer, Heidelberg (2005)
6. Butenhof, D.R.: Programming with POSIX threads. Addison-Wesley Longman Publishing Co., Inc., Boston (1997)
7. Cohen, H.: A Course in Computational Algebraic Number Theory. Undergraduate Texts in Mathematics. Springer, Heidelberg (1993)
8. Coster, M., Joux, A., LaMacchia, B., Odlyzko, A., Schnorr, C., Stern, J.: Improved Low-Density Subset Sum Algorithm. *Journal of Computational Complexity* 2, 111–128 (1992)
9. Coster, M., LaMacchia, B., Odlyzko, A., Schnorr, C.: An Improved Low-Density Subset Sum Algorithm. In: Davies, D.W. (ed.) EUROCRYPT 1991. LNCS, vol. 547, pp. 54–67. Springer, Heidelberg (1991)
10. Damgård, I.: A design principle for hash functions. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 416–427. Springer, Heidelberg (1990)
11. Filipovic, B.: Implementierung der Gitterbasenreduktion in Segmenten. Master's thesis, University of Frankfurt am Main (2002)
12. Goldstein, A., Mayer, A.: On the equidistribution of Hecke points. In: Formum Mathematicum, vol. 15, pp. 165–189 (2003)
13. Heckler, C.: Automatische Parallelisierung und parallele Gitterbasisreduktion. PhD thesis, Universität des Saarlandes, Saarbrücken (1995)
14. Heckler, C., Thiele, L.: Parallel complexity of lattice basis reduction and a floating-point parallel algorithm. In: Reeve, M., Bode, A., Wolf, G. (eds.) PARLE 1993. LNCS, vol. 694, pp. 744–747. Springer, Heidelberg (1993)
15. Heckler, C., Thiele, L.: A parallel lattice basis reduction for mesh-connected processor arrays and parallel complexity. In: Proceedings of SPDP 1993, Dallas, pp. 400–407 (1993)
16. Joux, A.: A Fast Parallel Lattice Basis Reduction Algorithm. In: Proceedings of the Second Gauss Symposium, pp. 1–15 (1993)
17. Joux, A.: La Réduction des Réseaux en Cryptographie. PhD thesis, Laboratoire d'Informatique de L'Ecole Normale Supérieure LIENS, Paris, France (1993)
18. Koy, H., Schnorr, C.: Segment LLL-Reduction with Floating Point Orthogonalization. In: Silverman, J.H. (ed.) CaLC 2001. LNCS, vol. 2146, pp. 81–96. Springer, Heidelberg (2001)
19. Lagarias, C., Odlyzko, A.: Solving Low-Density Subset Sum Problems. *JACM* 32, 229–246 (1985)
20. Lenstra, A., Lenstra, H., Lovász, L.: Factoring Polynomials with Rational Coefficients. *Math. Ann.* 261, 515–534 (1982)
21. Louis Roch, J.-L., Villard, G.: Parallel gcd and lattice basis reduction. In: CONPAR 1992 and VAPP 1992. LNCS, vol. 634, pp. 557–564. Springer, Heidelberg (1992)
22. May, A.: New RSA Vulnerabilities Using Lattice Reduction Methods. PhD thesis, University of Paderborn (2003)
23. May, A.: Secret Exponent Attacks on RSA-type Schemes with Moduli $n=pq$. In: Bao, F., Deng, R., Zhou, J. (eds.) PKC 2004. LNCS, vol. 2947, pp. 218–230. Springer, Heidelberg (2004)
24. Micciancio, D., Goldwasser, S.: Complexity of Lattice Problems—A Cryptographic Perspective. Kluwer Academic Publishers, Dordrecht (2002)

25. Nguyn, P.Q., Stehl, D.: Low-dimensional lattice basis reduction revisited. In: Buell, D.A. (ed.) ANTS 2004. LNCS, vol. 3076, pp. 338–357. Springer, Heidelberg (2004)
26. Nguyn, P.Q., Stehl, D.: Floating-point LLL revisited. In: Cramer, R. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 215–233. Springer, Heidelberg (2005)
27. Nguyn, P.Q., Stehl, D.: LLL on the average. In: Hess, F., Pauli, S., Pohst, M. (eds.) ANTS 2006. LNCS, vol. 4076, pp. 238–256. Springer, Heidelberg (2006)
28. Nguyen, P., Stern, J.: Adapting Density Attacks to Low-Weight Knapsacks. In: Roy, B. (ed.) ASIACRYPT 2005. LNCS, vol. 3788, pp. 41–58. Springer, Heidelberg (2005)
29. Nguyen, P.Q.: Cryptanalysis of the Goldreich-Goldwasser-Halevi Cryptosystem from Crypto 1997. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 288–304. Springer, Heidelberg (1999)
30. Nguyen, P.Q., Stern, J.: Lattice Reduction in Cryptology: An Update. In: Bosma, W. (ed.) ANTS 2000. LNCS, vol. 1838, pp. 85–112. Springer, Heidelberg (2000)
31. Pohst, M.E., Zassenhaus, H.: Algorithmic Algebraic Number Theory. Cambridge University Press, Cambridge (1989)
32. Regev, O.: Lattice-based cryptography. In: Dwork, C. (ed.) CRYPTO 2006. LNCS, vol. 4117, pp. 131–141. Springer, Heidelberg (2006)
33. Schnorr, C., Euchner, M.: Lattice Basis Reduction: Improved Practical Algorithms and Solving Subset Sum Problems. In: Budach, L. (ed.) FCT 1991. LNCS, vol. 529, pp. 68–85. Springer, Heidelberg (1991)
34. Stevens, R.W., Rago, S.A.: Advanced Programming in the UNIX(R) Environment, 2nd edn. Addison-Wesley Professional, Reading (2005)
35. Villard, G.: Parallel lattice basis reduction. In: ISSAC 1992: Papers from the international symposium on Symbolic and algebraic computation, pp. 269–277. ACM, New York (1992)
36. Wetzel, S.: An Efficient Parallel Block-Reduction Algorithm. In: Buhler, J.P. (ed.) ANTS 1998. LNCS, vol. 1423, pp. 323–337. Springer, Heidelberg (1998)
37. Wetzel, S.: Lattice Basis Reduction Algorithms and their Applications. PhD thesis, Universitt des Saarlandes (1998)
38. Wiese, K.: Parallelisierung von LLL-Algorithmen zur Gitterbasisreduktion. Implementierung auf dem Intel iPSC/860 Hypercube. Master’s thesis, Universitt des Saarlandes (1994)

Efficient Parallel Implementation of Evolutionary Algorithms on GPGPU Cards

Ogier Maitre¹, Nicolas Lachiche¹, Philippe Clauss¹, Laurent Baumes²,
Avelino Corma², and Pierre Collet¹

¹ LSIIT University of Strasbourg, France

{maitre, lachiche, clauss, collet}@lsiit.u-strasbg.fr

² Instituto de Tecnología Química UPV-CSIC, Valencia Spain
{baumes1, acorma}@itq.upv.es

Abstract. A parallel solution to the implementation of evolutionary algorithms is proposed, where the most costly part of the whole evolutionary algorithm computations (the population evaluation), is deported to a GPGPU card. Experiments are presented for two benchmark examples on two models of GPGPU cards: first a "toy" problem is used to illustrate some noticeable behaviour characteristics before a real problem is tested out. Results show a speed-up of up to 100 times compared to an execution on a standard micro-processor. To our knowledge, this solution is the first showing such an efficiency with GPGPU cards. Finally, the EASEA language and its compiler are also extended to allow users to easily specify and generate efficient parallel implementations of evolutionary algorithms using GPGPU cards.

1 Introduction

Between nowadays available manycore architectures, GPGPU (General Purpose Graphic Processing Units) cards are offering one of the most attractive cost/performance ratio. However programming such machines is a difficult task. This paper focuses on a specific kind of resource-consuming application: evolutionary algorithms. It is well-known that such algorithms offer efficient solutions to many optimization problems, but they usually require a great number of evaluations, making processing power a limit on standard micro-processors. However, their algorithmic structure clearly exhibits resource-costly computation parts that can be naturally parallelized. But, GPGPU programming constraints induce to consider dedicated operations for efficient parallel execution, one of the main performance-relevant constraint being the time needed to transfer data from the host memory to the GPGPU memory.

This paper starts by presenting evolutionary algorithms, and studying them to determine where parallelization could take place. Then, GPGPU cards are presented in section 3, and a proposition on how evolutionary algorithms could be parallelized on such cards is made and described in section 4. Experiments are made on two benchmarks and two NVidia cards in section 5 and some related works are described in section 7. Finally, results and future developments are discussed in the conclusion.

2 Presentation of Evolutionary Algorithms

In [5], Darwin suggests that species evolve through two main principles: variation in the creation of new children (that are not exactly like their parents) and survival of the fittest, as many more individuals of each species are born than can possibly survive.

Evolutionary Algorithms (EAs) [9] get their inspiration from this paradigm to suggest a way to solve the following interesting question. Given :

1. a difficult problem for which no computable way of finding a good solution is known and where a solution is represented as a set of parameters,
2. a limited record of previous trials that have all been evaluated.

How can one use the accumulated knowledge to choose a new set of parameters to try out (and therefore do better than a random search) ? EAs rely on artificial Darwinism to do just that: create new potential solutions from variations on good individuals, and keeping a constant population size through selection of the best solutions. The Darwinian inspiration for this paradigm leads to borrow some specific vocabulary from biology: given an initial set of evaluated potential solutions (called a *population of individuals*), *parents* are *selected* among the best to create *children* thanks to *genetic operators* (that Darwin called “variation” operators), such as *crossover* and *mutation*. *Children* (new potential solutions) are then evaluated and from the pool of *parents* and *children*, a *replacement* operator selects those that will make it to the new *generation* before the loop is started again.

2.1 Parallelization of a Generic Evolutionary Algorithm

The algorithm presented on figure II contains several steps that may or may not be independent. To start with, population initialisation is inherently parallel, because all individuals are created independently (usually with random values).

Then, all newly created individuals need to be evaluated. But since they are all evaluated independently using a *fitness* function, evaluation of the population can be done in parallel. It is interesting to note that in evolutionary algorithms, evaluation of individuals is usually the most CPU-consuming step of the algorithm, due to the high complexity of the fitness function.

Once a parent population has been obtained (by evaluating all the individuals of the initial population), one needs to create a new population of children. In order to create a child, it is necessary to select some parents on which variation operators (crossover, mutation) will be applied. In evolutionary algorithms, selection of parents is also parallelizable because one parent can be selected several times, meaning that independent selectors can select whoever they wish without any restrictions.

Creation of a child out of the selected parents is also a totally independent step: a crossover operator needs to be called on the parents, followed by a mutation operator on the created child.

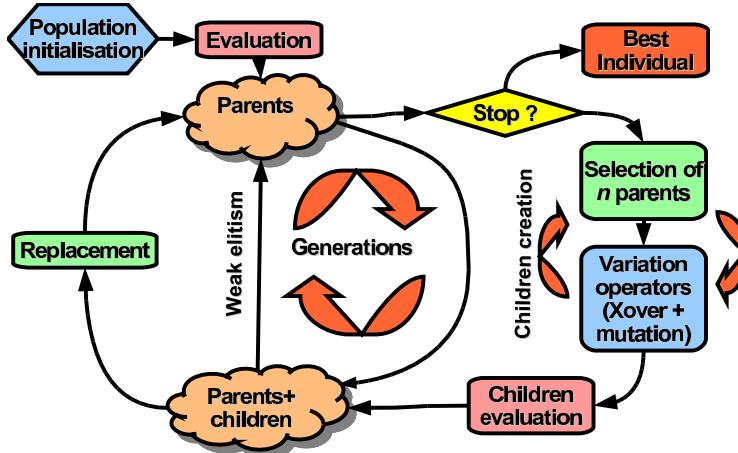


Fig. 1. Generic evolutionary loop

So up to now, all steps of the evolutionary loop are inherently parallel but for the last one: replacement. In order to preserve diversity in the successive generations, the $(N + 1)$ -th generation is created by selecting some of the best individuals of the parents+children populations of generation N . However, if an individual is allowed to appear several times in the new generation, it could rapidly become preeminent in the population, therefore inducing a loss of diversity that would reduce the exploratory power of the algorithm.

Therefore, evolutionary algorithms impose that all individuals of the new generation are different. This is a real restriction on parallelism, since it means that the selection of N survivors cannot be made independently, otherwise a same individual could be selected several times by several independent selectors.

Finally, one could wonder whether several generations could evolve in parallel. The fact that generation $(N + 1)$ is based on generation N invalidates this idea.

3 GPGPU Architecture

GPGPU and classic CPU designs are very different. GPGPUs come from the gaming industry and are designed to do 3D rendering. They inherit specific features from this usage. For example, they feature several hundreds execution units grouped into SIMD bundles that have access to a small amount of shared memory (16KB on the NVidia 8800GTX that was used for this paper), a large memory space (several hundred megabytes), a special access mode for texture memory and a hardware scheduling mechanism.

The 8800GTX GPGPU card features 128 stream processors (compared to 4 general purpose processors on the Intel Quad Core) even though both chips share a similar number of transistors (681 million for the 8800GTX *vs* 582 million for the Intel Quad Core). This can be done thanks to a simplified architecture that has some serious drawbacks. For instance, all stream processors are not

independent. They are grouped into SIMD bundles (16 SPMD bundles of 8 SIMD units on the 8800GTX, which saves 7 fetch and dispatch units). Then, space-consuming cache memory is simply not available on GPGPUs, meaning that all memory accesses (that can be done in only a few cycles on a CPU if the data is already in the cache) cost several hundred cycles.

Fortunately, some workarounds are provided. For instance, the hardware scheduling mechanism allows to run a bundle of threads called a *warp* at the same time, swapping between the warps as soon as a thread of the current warp is stalled on a memory access, so memory latency can be overcome with warp scheduling. But there is a limit to what can be done: it is important to have enough parallel tasks to be scheduled while waiting for the memory. A thread's state is not saved into memory. It stays on the execution unit (like in hyper-threading mechanism), so the number of registers used by a task directly impacts the number of tasks that can be scheduled on a bundle of stream processors. Then, there is a limit in the number of schedulable warps (24 warps, i.e. 768 threads on the 8800GTX).

All these quirks make it very difficult for standard programs to exploit the real power of these graphic cards.

4 Parallel Implementation on a GPGPU Card

As has been shown in section 2.1, it is possible to parallelize most of the evolutionary loop. However, whether it is worth to run everything in parallel on the GPGPU card or not is another matter: in [87][1], the authors implemented complete algorithms on GPGPU cards, but clearly show that doing so is very difficult, for quite few performance gains.

Rather than going this way, the choice made for this paper was to keep everything simple, and start with experimenting the obvious idea of only parallelizing children evaluation, based on the three following considerations.

1. Implementing the complete evolutionary engine on the GPGPU card is very complex, so it seems preferable to start with parallelizing only one part of the algorithm.
2. Usually, in evolutionary algorithms, execution of the evolutionary engine (selection of parents, creation of children, replacement step) is extremely fast compared to the evaluation of the population.
3. Then, if the evolutionary engine is kept on the host CPU, one needs to transfer the genome of the individuals only once on the GPGPU for each generation. If the selection and variation operators (crossover, mutation) had been implemented on the GPGPU, it would have been necessary to get the population back on the host CPU at every generation for the replacement step.

Evaluation of the population on the GPGPU is a massively parallel process that suits well an SPMD/SIMD computing model, because standard evolutionary algorithms use the same evaluation function to evaluate all individuals¹.

¹ This is not the case in Genetic Programming, where on the opposite, all individuals are different functions that are tested on a common data learning set.

Individual evaluations are grouped into structures called Blocks, that implement a group of threads which can be executed on a same bundle. Dispatching individuals evaluation across this structure is very important in order to maximize the load on the whole GPGPU. Indeed, as seen in section 3 a bundle has limited scheduling capacity, depending on the hardware scheduling device or register limitations. The GPGPU computing unit must have enough registers to execute all the tasks of a block at the same time, representing the scheduling limit. The 8800GTX card has a scheduling capacity of 768 threads, and 8192 registers. So one must make sure that the number of individuals in a block is not greater than the scheduling capacity, and that there are enough individuals on a bundle in order to maximize this capacity. In this paper, the implemented algorithm spreads the population into $n * k$ blocks, where n is the number of bundles on the GPGPU and k is the integer ceiling of $\frac{popSize}{n*schedLimit}$. This simple algorithm yields good results in tested cases. However, a strategy to automatically adapt blocks definitions to computation complexity either by using a static or dynamic approach needs to be investigated in some future work.

When a population of children is ready to be evaluated, it is copied onto the GPGPU memory. All the individuals are evaluated with the same evaluation function, and results (fitnesses) are sent back to the host CPU, that contains the original individuals and manages the populations.

On a standard host CPU EA implementation, an individual is made of a genome plus other information, such as its fitness, and statistics information (whether it has recently been evaluated or not, . . .). So the transfer of n genomes only onto the GPGPU would result in n transfers and individuals scattered everywhere in GPGPU memory. Such a number of memory transfers would have been unacceptable. So, it was chosen to ensure spatial locality of all genomes to be contiguous into host CPU memory before the transfer, by copying every newly created individual in a buffer right after its creation. Then this buffer is sent to the GPU memory in one single transfer. Some experiments showed that on a particular case, with a large number of children, the transfer time went from 80 seconds with scattered data down to 180 μ s with a buffer of contiguous data.

Our implementation uses only global memory since in the general case, the evaluation function does not generate significant data reuse that would justify the use of the small 16KB shared memory or the texture cache. Indeed with shared memory, the time saved in data accesses is generally wasted by data transfers between global memory and shared memory. Notice that shared memory is not accessible from the host CPU part of the algorithm. Hence one has to first copy data into global memory, and in a second step into shared memory.

The chosen implementation strategy exhibits the main overhead risk as being the time spent to transfer the population onto the GPGPU memory. Hence the (computation time)/(transfer time) ratio needs to be large enough to effectively take advantage of the GPGPU card. Experiments on data transfer rate show that on the 8800GTX, a 500 MB/s bandwidth was reached, which is much lower than the advertised 4GB/s maximum bandwidth. However, experiments presented in the following show that this rate is quite acceptable even for very simple evaluation functions.

5 Experiments

Two implementations have been tested: a toy problem that contains interesting tuneable parameters allowing to observe the behaviour of the GPGPU card, and a much more complex real world problem to make sure that the GPGPU processors are also able to run more complex fitness functions. In fact, the 400 code lines of the real world evaluation function were programmed by a chemist, who has not the least idea on how to use a GPGPU card.

5.1 The Weierstrass Benchmark Program

Weierstrass-Mandelbrot test functions, defined as:

$$W_{b,h}(x) = \sum_{i=1}^{\infty} b^{-ih} \sin(b^i x) \text{ with } b > 1 \text{ and } 0 < h < 1$$

are very interesting to use as a test case of CPU usage in evolutionary computation since they provide two parameters that can be adjusted independently.

Theory defines Weierstrass functions as an infinite sum of sines. Programmers perform a finite number of iterations to compute an approximation of the function. The number of iterations is closely related to the host CPU time spent in the evaluation function.

Another parameter that can also be adjusted is the dimension of the problem: a 1,000 dimension Weierstrass problem takes 1,000 continuous parameters, meaning that its genome is an array of 1,000 float values, while a 10 dimension problem only takes 10 floats. The 10 dimension problem will evidently take much less time to evaluate than the 1,000 dimension problem. But since evaluation time also depends on the number of iterations, tuning both parameters provides many configurations combining genome size and evaluation time.

Figure 2 left shows the time taken by the evolutionary algorithm to compute 10 generations on both a 3.6GHz Pentium computer and an 8800GTX GPGPU

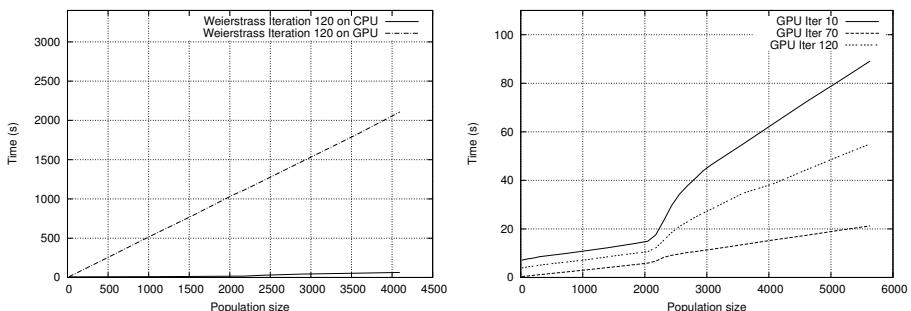


Fig. 2. Left: Host CPU (top) and CPU+8800GTX (bottom) time for 10 generations of the Weierstrass problem on an increasing population size. Right: CPU+8800GTX curve only, for increasing numbers of iterations and increasing population sizes.

card, for 1000 dimensions, 120 iterations and a number of evaluations per generation growing from 16 to 4,096 individuals (number of children = 100% of the population). This represents the total time (including what is serially done on the host CPU (population management, crossovers, mutations, selections, …) on both architectures (host CPU only and host CPU+GPU).

For 4,096 evaluations ($\times 10$ generations), the host CPU spends 2,100 seconds while the host CPU and 8800GTX only spends 63 seconds, resulting in a speedup of 33.3.

Figure 2 right shows the same GPGPU curve, for different iteration counts. On this second figure, one can see that the 8800GTX card steadily takes in more individuals to evaluate in parallel without much difference in evaluation time until the threshold number of 2048 individuals is reached, after which it gets saturated. Beyond this value, evaluation time increases linearly with the number of individuals, which is normal since the parallel card is already working on a full load. It is interesting to see that with 10 iterations, the curve before and after 2048 has nearly the same slope, meaning that for 10 iterations, the time spent in the evaluation function is negligible, so the curve mainly shows the overhead time.

Since using a GPGPU card induces a necessary overhead, it is interesting to determine when it is advantageous to use an 8800GTX card. Figure 3 left shows that on a small problem with 10 dimensions Weierstrass and 10 iterations running in virtually no time, this threshold is met between 400 individuals or 600 individuals, depending whether the genome size uses 40 bytes or 4 kilobytes, which is quite a big genome.

The steady line (representing the host CPU) shows an evaluation time slightly shorter than 0.035 milliseconds, which is very short, even on a 3.6GHz computer.

The 3 GPGPU curves show that indeed, the size of the genomes has an impact when individuals are passed to the GPGPU card for evaluations. On this figure, evaluation is done on a 10 dimension Weierstrass function corresponding to a 40 bytes size (the 8800GTX card only accepts floats). The additional genome data is not used on the 2 kilobytes and 4 kilobytes genomes, in order to isolate the time taken to transfer large size genomes to the GPGPU for all the population.

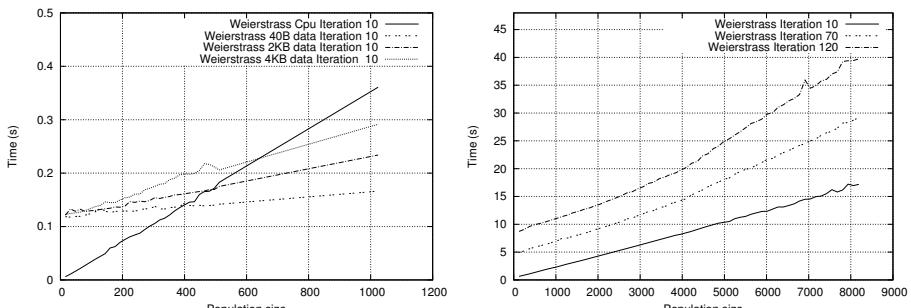


Fig. 3. Left: determination of genome size overhead on a very short evaluation. Right: Same curves as figure 2 right, but for the GTX260 card.

On figure 3 right, the same measures are shown with a recently acquired GTX260 NVidia card. One can see that with this card, total time is only 20 seconds for a population of 5,000 individuals while the 8800GTX card takes 60 seconds and the 3.6GHz Pentium takes 2,100 seconds. So where the 8800GTX *vs* host CPU speedup was 33.3, the GTX260 *vs* host CPU speedup is about 105, which is quite impressive for a card that only costs around \$250.00.

5.2 Application to a Real-World Problem

In materials science, knowledge of the structure at an atomistic/molecular level is required for any advanced understanding of its performance, due to the intrinsic link between the structure of the material and its useful properties. It is therefore essential that methods to study structures are developed.

Rietveld refinement techniques [10] can be used to extract structural details from an X-Ray powder Diffraction (XRD) pattern [214], provided an approximate structure is known. However, if a structural model is not available, its determination from powder diffraction data is a non-trivial problem. The structural information contained in the diffracted intensities is obscured by systematic or accidental overlap of reflections in the powder pattern.

As a consequence, the application of structure determination techniques which are very successful for single crystal data (primarily direct methods) is, in general, limited to simple structures. Here, we focus on inherently complex structures of a special type of crystalline materials whose periodic structure is a 4-connected 3 dimensional net such as alumino-silicates, silico-alumino-phosphates (SAPO), alumino-phosphates (AlPO), etc...

The genetic algorithm is employed in order to find “correct” locations, e.g. from a connectivity point of view, of T atoms. As the distance $T - T$ for bonded atoms lies in a fixed range [dmin-dmax], the connectivity of each new configurations of T atoms can be evaluated. The fitness function corresponds to the number of defects in the structure, and Fitness= $f_1 + f_2$ is defined as follows:

1. All T atoms should be linked to 4 and only 4 neighbouring T s, so:

$$f_1 = \text{Abs}(4 - \text{Number_of_Neighbours})$$

2. no T should be too close, e.g. $T - T < dmin$, so:

$$f_2 = \text{Number_of_Too_Close_T_Atoms}$$

Speedup on this chemical problem: As mentioned earlier, the source code came from our chemist co-author, who is not a programming expert (but is nevertheless capable of creating some very complex code) and knows nothing about GPGPU architecture and its use.

First, while the 3.60GHz CPU was evaluating 20,000 individuals in 23 seconds only, which seemed really fast considering the very complex evaluation function, the GPGPU version took around 80 seconds which was disappointing.

When looking at the genome of the individuals, it appeared that it was coded in a strange structure, i.e. an array of 4 pointers towards 4 other arrays of 3

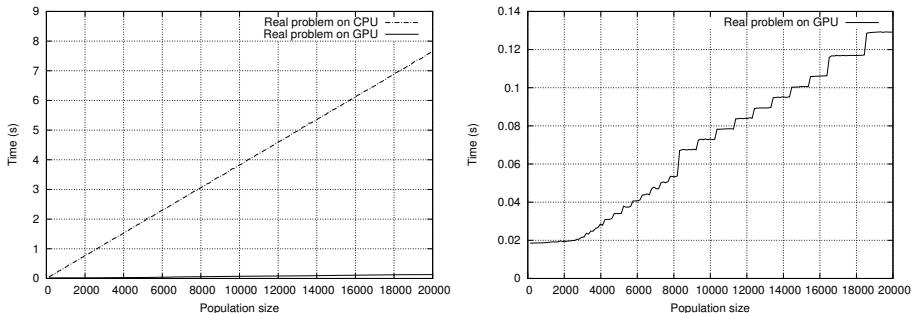


Fig. 4. Left: evaluation times for increasing population sizes on host CPU (top) and host CPU + GTX260 (bottom). Right: CPU + GTX260 total time.

floats. This structure seemed much too complex to access, so it was suggested to flatten it into a unique array of 12 floats which was easy to do, but unfortunately, the whole evaluation function was made of pointers to some parts of the previous genome structure. After some hard work, all got back to a pointer-less flat code, and evaluation time for the 20,000 individuals instantaneously dropped from 80 second down to 0.13 seconds. One conclusion to draw out of this experience is that, as expected, GPGPUs do not seem very talented in allocating, copying and de-allocating memory.

Back to the host CPU, the new function now took 7.66s to evaluate 20,000 individuals, meaning that all in all, the speedup offered by the GPGPU card is nearly 60 on the new GTX260 (figure 4 left). Figure 4 right shows only the GTX260 curve.

6 EASEA: Evolutionary Algorithm Specification Language

EASEA² [3] is software platform that was originally designed to help non expert programmers to try out evolutionary algorithms to optimise their applied problems without the need to implement a complete algorithm. It was revived to integrate the parallelization of children evaluation on NVidia GPGPU cards.

EASEA is a specification language that comes with its dedicated compiler. Out of the `weierstrass.ez` specification program shown in fig. 5, the EASEA compiler will output a C++ source file that implements a complete evolutionary algorithm to minimize the Weierstrass function.

The `-cuda` option has been added to the compiler in order to automatically output a source code that will parallelize the evaluation of the children population on a CUDA-compatible NVidia card out of the same `.ez` specification code, therefore allowing anyone who wishes to use GPGPU cards to do so without any knowledge about GPGPU programming. However, a couple of guidelines that were included in the latest version of the EASEA user manual, should be

² EASEA (pron. [i:zi:]) stands for EASty Specification for Evolutionary Algorithms.

```

1  \User classes :
2  GenomeClass {
3      float x[SIZE];
4  }
5
6  \GenomeClass::initialiser :
7  for (int i=0; i<N; i++) {
8      Genome.x[i] = random
9          (-1.0,1.0);
10
11 \GenomeClass::mutator :
12 for (int i=0; i<N; i++) {
13     if (tossCoin(pMutProb)){
14         Genome.x[i]+=SIGMA*
15             random(0.,1.);
16         Genome.x[i]=MAX(X_MIN,
17             MIN(X_MAX,Genome.x[i
18                 ]));
19     }
20
21 \GenomeClass::crossover:
22 for(i=0; i<N; i++){
23     float a = (float)randomLoc
24         (0.,1.);
25     if (&child1)
26         child1.x[i] =
27             a*parent1.x[i]+
28             (1-a)*parent2.x[i];
29 }
30
31 \GenomeClass::evaluator:
32 float res = 0., b=2.;
33 float h=.25, val[SIZE];
34 int i,k;
35 for(i = 0;i<N; i++){
36     val[i] = 0.;
37     for (k=0;k<ITER;k++)
38         val[i] +=
39             pow(b,-(float)k*h) *
40                 sin(pow(b,(float)k)*
41                     Genome.x[i]);
42     res += Abs(val[i]);
43 }
44
45 return (res);
46 }
```

Fig. 5. Evolutionary algorithm minimizing the Weierstrass function in EASEA

followed, such as using flat genomes rather than pointers towards matrices, fitness functions that do not use more registers than available on a GPGPU unit, floats rather than doubles³ ...

Adding the `-cuda` option to this compiler is very important, since it not only allows replication of the presented work, but also gives non GPGPU expert programmers the possibility to run their own code on these powerful parallel cards.

7 Related Work

Even though many papers have been written on the implementation of Genetic Programming algorithms on GPGPU cards, only three papers were found on the implementation of standard evolutionary algorithms on these cards.

In [11], Yu *et al.* implement a refined fine-grained algorithm with a 2D toroidal population structure stored as a set of 2D textures, which imposes restrictions on mating individuals (that must be neighbours). Other constraints arise, such as the need to store a matrix of random numbers in GPGPU memory for future

³ Although available on all recent GPGPU cards, using double precision variables will apparently considerably slow down the calculations on current GPGPU cards, but this has not been tested yet, as all this work has been done on an 8800GTX card that can only manipulate floats.

reference, since there is no random number generator on the card. Anyway, a 10 time speedup is obtained, but on a huge population of 512×512 individuals.

In [7], Fok *et al.* find that standard genetic algorithms are ill-suited to run on GPGPUs because of such operators as crossovers “that would slow down execution when executed on the GPGPU” and therefore choose to implement a crossover-less Evolutionary Programming algorithm [6] here again entirely on the GPGPU card. The obtained speedup of their parallel EP “ranges from 1.25 to 5.02 when the population size is large enough.”

In [8], Li *et al.* implement a *Fine Grained Parallel Genetic Algorithm* once again on the GPGPU, to “avoid massive data transfer.” For a strange reason, they implement a *binary* genetic algorithm even though GPGPUs have no bit-wise operators, so go into a lot of trouble to implement simple genetic operators.

To our knowledge no paper proposed the simple approach of only parallelizing the evaluation of the population on the GPGPU card.

8 Conclusion and Future Developments

Results show that deporting the children population onto the GPGPU for a parallel evaluation yields quite significant speedups of up to 100 on a \$250 GTX260 card, in spite of the overhead induced by the population transfer.

Being faster by around 2 orders of magnitude is a real breakthrough in evolutionary computation, as it will allow applied scientists to find new results in their domains. Then, researchers on artificial evolution will need to modify their algorithms to adapt them to such speeds, that will probably lead to premature convergence, for instance. Then, unlike many other works that are difficult (if not impossible) to replicate, knowhow on the parallelization of evolutionary algorithms has been integrated into the EASEA language. Researchers who would like to try out these cards can simply specify their algorithm using EASEA and the compiler will parallelize the evaluation.

Anyway, many improvements can still be expected. Load balancing could probably be improved, in order to maximize bundles throughput. Using texture cache memory may be interesting on evaluation functions that repeatedly access genome data. Automatic use of shared memory could also yield some good results, particularly on local variables in the evaluation function.

Finally, an attempt to implement evolutionary algorithms on Sony/Toshiba/IBM Cell multicore chips is currently being made. Its integration into the EASEA language could allow to compare performance of GPGPU and Cell architecture on identical programs.

References

1. Baumes, L.A., Moliner, M., Corma, A.: Design of a full-profile matching solution for high-throughput analysis of multi-phases samples through powder x-ray diffraction. *Chemistry - A European Journal* (in press)
2. Baumes, L.A., Moliner, M., Nicoloyannis, N., Corma, A.: A reliable methodology for high throughput identification of a mixture of crystallographic phases from powder x-ray diffraction data. *CrystEngComm.* 10, 1321–1324 (2008)

3. Collet, P., Lutton, E., Schoenauer, M., Louchet, J.: Take it easy. In: *Informatics: 10 Years Back, 10 Years Ahead*. LNCS, pp. 891–901. Springer, Heidelberg (2000)
4. Corma, A., Moliner, M., Serra, J.M., Serna, P., Diaz-Cabanas, M.J., Baumes, L.A.: A new mapping/exploration approach for the synthesis of zeolites. *Chemistry of Materials*, 3287–3296 (2006)
5. Darwin, C.: *On the Origin of Species by Means of Natural Selection or the Preservation of Favoured Races in the Struggle for Life*. John Murray, London (1859)
6. Fogel, D.B.: Evolving artificial intelligence. Technical report (1992)
7. Fok, K.-L., Wong, T.-T., Wong, M.-L.: Evolutionary computing on consumer graphics hardware. *IEEE Intelligent Systems* 22(2), 69–78 (2007)
8. Li, J.-M., Wang, X.-J., He, R.-S., Chi, Z.-X.: An efficient fine-grained parallel genetic algorithm based on gpu-accelerated. In: *NPC Workshops. IFIP International Conference on Network and Parallel Computing Workshops, 2007*, pp. 855–862 (2007)
9. De Jong, K.: *Evolutionary Computation: a Unified Approach*. MIT Press, Cambridge (2005)
10. Young, R.A.: *The Rietveld Method*. OUP and International Union of Crystallography (1993)
11. Yu, Q., Chen, C., Pan, Z.: Parallel genetic algorithms on programmable graphics hardware. In: Wang, L., Chen, K., S. Ong, Y. (eds.) *ICNC 2005*. LNCS, vol. 3612, pp. 1051–1059. Springer, Heidelberg (2005)

Topic 12

Theory and Algorithms for Parallel Computation

Introduction

Andrea Pietracaprina*, Rob Bisseling*, Emmanuelle Lebhar*,
and Alexander Tiskin*

Parallelism has been traditionally at the base of high-end complex platforms. In recent years, however, the emergence of novel technologies and computing systems has broadened considerably the spectrum of applications where parallel computation plays a fundamental role: on the one hand, chip manufacturers are shifting their focus towards new designs incorporating multiple cores within individual chips; on the other, large-scale networks are becoming the underlying infrastructures of innovative computing and data access platforms. Both traditional and emerging scenarios pose a number of challenging theoretical questions regarding architectures, models, and algorithmic strategies for effective parallel computing. Topic 12 is a suitable venue where researchers can discuss these questions and present their new achievements.

This year, seven papers were submitted to the topic, targeting a wide range of problems. After a thorough revision process, two papers were accepted for presentation at the conference, resulting in about a 29% acceptance rate.

The first paper, *Implementing Parallel Google Map-Reduce in Eden*, by J. Berthold, M. Dieterle, and R. Loogen, discusses efficient parallel implementations of the Google map-reduce paradigm in Eden, a parallel extension of the Haskell language, and reports experimental results on some case studies. The paper offers also a critical perspective on the generality of the paradigm as a programming model. The second paper, *A Lower Bound for Oblivious Dimensional Routing*, by A. Osterloh, studies a fundamental packet routing primitive, namely k - k routing, on multidimensional meshes, proving a tight (asymptotic) lower bound on the number of steps it requires when the number d of dimensions is odd and when routing paths are restricted to traverse edges of any given dimension, consecutively. The result generalizes a known lower bound for $d = 3$.

* Topic chairs.

Implementing Parallel Google Map-Reduce in Eden

Jost Berthold, Mischa Dieterle, and Rita Loogen

Philipps-Universität Marburg, Fachbereich Mathematik und Informatik

Hans Meerwein Straße, D-35032 Marburg, Germany

{berthold,dieterle,loogen}@informatik.uni-marburg.de

Abstract. Recent publications have emphasised map-reduce as a general programming model (labelled Google map-reduce), and described existing high-performance implementations for large data sets. We present two parallel implementations for this Google map-reduce skeleton, one following earlier work, and one optimised version, in the parallel Haskell extension Eden. Eden’s specific features, like lazy stream processing, dynamic reply channels, and nondeterministic stream merging, support the efficient implementation of the complex coordination structure of this skeleton. We compare the two implementations of the Google map-reduce skeleton in usage and performance, and deliver runtime analyses for example applications. Although very flexible, the Google map-reduce skeleton is often too general, and typical examples reveal a better runtime behaviour using alternative skeletons.

1 Introduction

To supply conceptual understanding and abstractions of parallel programming, the notion of *algorithmic skeletons* has been coined by Murray Cole in 1989 [1]. An algorithmic skeleton abstractly describes the (parallelisable) structure of an algorithm, but separates specification of the concrete work to do as a parameter function. Skeletons are meant to offer ready-made efficient implementations for common algorithmic patterns, the specification of which remains sequential. Thus, an algorithmic skeleton contains inherent parallelisation potential in the algorithm, but this remains hidden in its implementation. A broader research community has quickly adopted and developed the idea further [2]. Skeletons are a well-established research subject in the scientific community, yet until today they have only little impact on mainstream software engineering, in comparison with other models, like MPI [3] collective operations and *design patterns* [4].

To a certain extent, MPI collective operations and algorithmic skeletons follow the same philosophy: to specify common patterns found in many applications, and to provide optimised implementations that remain hidden in libraries. However, while skeletons describe a whole, potentially complex, algorithm, collective operations only predefine and optimise common standard tasks which are often needed in implementing more complex algorithms. The design pattern paradigm

has considerable potential in identifying inherent parallelism in common applications, capturing complex algorithmic structures, and providing conceptual insight in parallelisation techniques and problem decomposition. However, it often merely targets concepts and leaves implementation to established low-level libraries and languages (see e.g. textbook [5] for a typical example). Design patterns thus cannot provide the programming comfort and abstraction level of algorithmic skeletons. Moreover, collective operations and design patterns, by their very nature, are explicit about parallelism already in their specification, whereas skeletons completely hide parallelism issues.

Even more remarkable is the fact that applications from industry have meanwhile achieved the mainstream breakthrough for the skeleton idea (even though it is never called like this). In 2004, we saw the first publication which abstractly described large-scale *map-and-reduce* data processing at Google [6,7]. It was proposed as a “programming model” for large dataset processing, but in fact precisely realises the skeleton idea. A publication by Ralf Lämmel [8] points out shortcomings of the skeleton’s formal specification, provides sequential Haskell implementations, and briefly discusses parallelism. Given the great acceptance that the programming model has found, and its close relation to skeleton programming, this paper investigates possible parallel implementations starting from Lämmel’s Haskell code, and discusses their respective advantages and drawbacks.

From the perspective of functional languages, skeletons are specialised higher-order functions with a parallel implementation. Essentially, the skeleton idea applies a functional paradigm for coordination, independent of the underlying *computation language*. While skeleton libraries for imperative language, e.g. [9,10], typically offer a fixed, established set of skeletons, parallel functional languages are able to express new skeletons, or to easily create them by composition [11,12]. Some functional languages parallelise by pre-defined data parallel operations and skeletons, like NESL [12], OCamlP31 [13], or PMLS [11]. These fixed skeleton implementations are highly optimised and allow composition, but not the definition of new problem-specific skeletons or operations. More explicit functional coordination languages are appropriate tools not only to apply skeletons, but also for their *implementation*, allowing formal analysis and conceptual modeling. Coordination structure, programming model and algorithm structure can be cleanly separated by functional languages, profiting from their abstract, mathematically oriented nature. In our work, we use the general-purpose parallel Haskell dialect Eden [14] as an implementation language for the skeletons.

The paper is structured as follows: Section 2 explains the classical map-and-reduce skeleton defined in Eden, thereby introducing features of the language we use for our implementations. Section 3 introduces the Google map-reduce skeleton. Parallel implementation variants are discussed in Section 4. A section with measurements and analyses for some example applications follows. Section 6 considers related work, the final section concludes.

2 Parallel Transformation and Reduction in Eden

Classically, reduction over a list of elements is known as the higher-order function `fold` (from the left or from the right), and is often combined with a preceding transformation of the list elements (in other words, `map`). Denotationally, it is a composition of the higher-order functions `map` and `fold`:

```
mapFoldL :: (a -> b) -> (c -> b -> c) -> c -> [a] -> c
mapFoldL mapF redF n list = foldl redF n (map mapF list)
```

```
mapFoldR :: (a -> b) -> (b -> c -> c) -> c -> [a] -> c
mapFoldR mapF redF n list = foldr redF n (map mapF list)
```

In this general form, the folding direction leads to slightly different types of the reduction operator `redF`. Parallel implementations have to unify types `b` and `c` and require associativity to separate sub-reductions. In addition, the parameter `n` should be neutral element of `redF`. Under these conditions, the folding direction is irrelevant, as both versions yield the same result. Parallel implementations may even reorder the input, requiring the reduction operator `redF` to be commutative.

Assuming associativity and commutativity, we can easily define a parallel map-and-reduce skeleton for input streams in the functional Haskell dialect Eden, as shown in Fig. 1.

```
parmapFoldL :: (Trans a, Trans b) =>
  Int ->          -- no. of processes
  (a -> b) ->    -- mapped on input
  (b -> b -> b) -> -- reduction (assumed commutative)
  b ->           -- neutral element for reduction
  [a] -> b

parmapFoldL np mapF redF neutral list = foldl' redF neutral subRs
  where sublists = unshuffle np list
        subFoldProc = process (foldl' redF neutral . (map mapF))
        subRs = spawn (replicate np subFoldProc) sublists

unshuffle :: Int -> [a] -> [[a]] -- distributes the input stream
unshuffle n list = ...           -- round-robin in np streams
spawn :: [Process a b] -> [a] -> [b] -- instantiates a set of processes
spawn ps inputs = ...           -- with respective inputs
```

Fig. 1. Parallel map-reduce implementation in Eden

The input stream is distributed round-robin into `np` inputs for `np` Eden processes, which are instantiated by the Eden library function `spawn`. `Process` is the type constructor for Eden Process abstractions, which are created by the function `process :: (a -> b) -> Process a b`. Type class `Trans` provides implicitly used data transfer functions. The spawned processes perform the transformation (`map`) and pre-reduce the map results (duplicating the given neutral element) by the *strict*

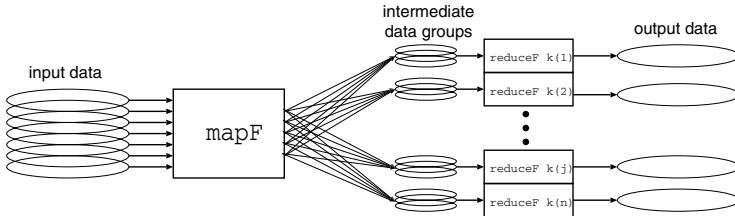


Fig. 2. Computation scheme of Google map-reduce

left-fold `foldl'`. As explained, the fold operator `redF` has got a restricted type in order to combine the subresults in a second stage, and is assumed commutative because input for the pre-reductions is distributed round-robin and streamed.

3 The “Google Map-Reduce” Skeleton

A more general variant of map-and-reduce has been proposed, as a programming model for processing large datasets, by Google personnel Jeff Dean and Sanjay Ghemawat. In January 2008, an update of the original publication (OSDI 2004 [6]) appeared in the ACM communications [7].

The intention to provide a framework which allows one “to express the simple computations [...] but hides the messy details of parallelization, fault-tolerance, data distribution, and load balancing, in a library” [6] is precisely the skeleton idea. However, the word “skeleton” does not figure in any of the two publications! Neither publication claims for the model to be new, its essential merit is that it brought the skeleton approach to industry. The model has found great acceptance as a programming model for parallel data processing (e.g. [15][6]).

The computation scheme of Google map-reduce is depicted in Fig. 2. In a nutshell, a Google map-reduce instance first transforms key/value pairs into (intermediate) other key/value pairs, using a `mapF` function. After this, each collection of intermediate data *with the same key* is reduced to one resulting key/value pair, using a `reduceF` function. In-between the transformation and the reduction, the intermediate data is grouped by keys, so the whole computation has two logical phases. The parallelisation described in the original work [6] imposes additional requirements on the applied parameter functions; which, on the other hand, have more liberal type constraints than what `map` and `foldl` would require. Ralf Lämmel, in his related publication [8], captures them in a formal specification derived from the original examples and description, using Haskell.¹

```
GOOGLE_MapReduce :: forall k1 k2 v1 v2 v3. Ord k2 => -- for grouping
  (k1 -> v1 -> [(k2,v2)]) -- 'map' function, with keys 1
  -> (k2 -> [v2] -> Maybe v3) -- 'reduce' function, can use key 2
  -> Map k1 v1 -- A key to input-value mapping
  -> Map k2 v3 -- A key to output-value mapping
```

¹ The code is provided online by Lämmel, so we do not reproduce it here, see <http://www.cs.vu.nl/~ralf/MapReduce/>

The ‘map’ function takes an input pair of type $(k1, v1)$ ($k1$ could actually be subsumed under $v1$ by pairing) and may produce a whole *list* of intermediate key-value pairs $[(k2, v2)]$ from it, which are then grouped by key (the ordering constraint enables more efficient data structures – an equality constraint `Eq k2` would suffice). Each of the value lists for a key is then processed by the ‘reduce’ function to yield a final result of type $v3$ for this key, or no output (thereby the `Maybe`-type). Unlike the usual `fold`, output does not necessarily have the same type as the intermediate value (but typically $v2$ and $v3$ are of the same type). So the parameter function of `fold` in the Google publications is not properly a function which could be the argument to a `fold` (i.e. `reduce`) operation, nor is it always a reduction in the narrow sense. Additionally, as Lämmel points out, the original paper confuses lists and sets: The input to a skeleton instance is neither a set, nor a list, but a finite mapping from keys to values, where duplicate values are not allowed for the same key. And likewise, the output of the skeleton conceptually does not allow the same key to appear twice.

Examples: A classical combination of `map` and `foldl` can be expressed as a special case of the more general skeleton. The `map` function here produces singleton lists and assigns a constant intermediate key 0 to every one. The reduction function ignores these keys, and left-folds the intermediate values as usual.

```
mapfold :: (a -> b) -> (b -> b -> b) -> b -> [a] -> b
mapfold mapF redF neutral input = head (map snd (toList gResult))
  where mapF' _ x = [(0, mapF x)]
        redF' _ list = Just (foldl' redF neutral list)
        gResult      = gOGLE_MapReduce mapF' redF'
                      (fromList (zip (repeat 0) input))
```

A more general example, often given in publications on Google map-reduce, is to compute how many times certain words appear in a collection of web pages. The input is a set of pairs: web page URLs and web page content (and the URL is completely ignored). The ‘map’ part retrieves all words from the content and uses them as intermediate keys, assigning constant 1 as intermediate value to all words. Reduction sums up all these ones to determine how many times a word has been found in the input.

```
wordOccurrence = gOGLE_MapReduce toMap forReduction
  where toMap      :: URL -> String -> [(String,Int)]
        toMap url content = zip (words content) (repeat 1)
        forReduction :: String -> [Int] -> Maybe Int
        forReduction word counts = Just (sum counts)
```

A range of other, more complex applications is possible, for instance, iteratively clustering large data sets by the k-means method, used as a benchmark in two recent publications [15][16]. We will discuss this benchmark in Section 5.

4 Parallel Google Map-Reduce

Google map-reduce offers different opportunities for parallel execution. First, it is clear that the `map` function can be applied to all input data independently.

```

-- inner interface
parMapReduce :: Ord k2 =>
  Int -> (k2 -> Int)           -- No. of partitions, key partitioning
  -> (k1 -> v1 -> [(k2,v2)]) -- 'map' function
  -> (k2 -> [v2] -> Maybe v3) -- 'combiner' function
  -> (k2 -> [v3] -> Maybe v4) -- 'reduce' function
  -> [Map k1 v1]               -- Distributed input data
  -> [Map k2 v4]               -- Distributed output data

parMapReduce parts keycode mAP cOMBINER rEDUCE
= map ( -- parallelise in n reducers: parmap
        reducePerKey rEDUCE      -- 7. Apply 'reduce' to each partition
        . mergeByKey )           -- 6. Merge scattered intermediate data
  . transpose                  -- 5. Transpose scattered partitions
  . map ( -- parallelise in n=m mappers: farm n
        map ( reducePerKey cOMBINER -- 4. Apply 'combiner' locally
              . groupByKey )      -- 3. Group local intermediate data
        . partition parts keycode -- 2. Partition local intermediate data
        . mapPerKey mAP )        -- 1. Apply 'map' locally to each piece

```

Fig. 3. Parallel Google map-reduce skeleton, following Lämmel [8]
(we have added the parallelisation annotations in bold face)

Furthermore, since reduction is done for every possible intermediate key, several processors can be used in parallel to reduce the values for different keys. Additionally, the mapper processes in the implementation perform pre-grouping of intermediate pairs by (a hash function of) intermediate keys. Usually, implementations strictly split the whole algorithm in two phases. The productive implementation described in [7] is based on intermediate files in Google's own shared file system GFS. Pre-grouped data is periodically written to disk, and later fetched and merged by the reducer tasks before they start reduction of values with the same key. This makes it possible to reassign jobs in case of machine failures, making the system more robust. Furthermore, at the end of the map phase, remaining map tasks are assigned to several machines simultaneously to compensate load imbalances.

Following Lämmel's specification. To enable parallel execution, Lämmel proposes the version shown in Fig. 3. Interface and functionality of the Google map-reduce skeleton are extended in two places.

First, input to the map function is grouped in bigger “map jobs”, which allows to adapt task size to the resources available. For instance, the job size can be chosen appropriately to fit the block size of the file system. For this purpose, the proposed outer interface (not shown) includes a size parameter and an estimation function. The skeleton input is sequentially traversed and partitioned into tasks with estimated size close to (but less than) the desired task size.

Second, *two* additional pre-groupings of equal keys are introduced, one for a pre-reduction in the mapper processes, and one to aggregate input for the reducer processes. The map operation receives a task (a set of input pairs), and

produces a varying number of intermediate output. This output is sorted using intermediate keys, and the map processes pre-group output with the same key, using the added parameter function `COMBINER`. In many cases, this combiner will be the same function as the one used for reduction, but in the general case, its type differs from the `REDUCE` function type. To reduce the (potentially unbounded) number of intermediate keys, these intermediate (pre-reduced) results are then partitioned into a fixed number of key groups for the reducer processes, using two additional skeleton parameters. The parameter `parts` indicates how many partitions (and parallel reducer processes) to use, and the function `keycode` maps (or: is *expected* to map; the code in [8] does not check this property) each possible intermediate key to a value between 1 and `parts`. This mimics the behaviour of the productive Google implementation, which saves partitioned data into n intermediate files per mapper.

Our parallel straightforward implementation of the skeleton consists of replacing the *map* calls in the code (see Fig. B) by appropriate parallel *map* implementations. The number of reducers, n , equals the number of `parts` into which the hash function `keycode` partitions the intermediate keys. A straightforward parallel *map* skeleton with one process per task can be used to create these n reducer processes. An implementation which verbally follows the description should create m mapper processes, which process a whole stream of input tasks each. Different Eden skeletons realise this functionality: a `farm` skeleton with static task distribution, or a dynamically balanced `workpool` [14]. However, the interface proposed by Lämmel lacks the `m` parameter, thus our parallelisation simply uses as many mappers as reducer processes, $n = m$.

An optimised implementation. A major drawback of this straightforward version, directly derived from Lämmel's code [8], is its strict partitioning into

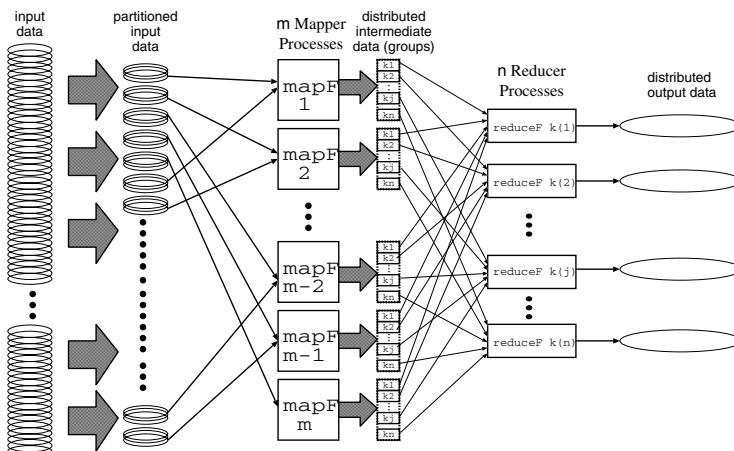


Fig. 4. Parallel Google map-reduce using distributed transpose functionality

the map phase and the reduce phase, and the call to `transpose` in between. In the Eden implementation suggested in Fig. 3, all intermediate data produced by the mapper processes is sent back to the caller, to be reordered (by `transpose`) and sent further on to the reducer processes.

Our optimised implementation uses direct stream communication between mappers and reducers, as depicted in Fig. 4. Furthermore, instances of mapper and reducer are gathered in one process, which saves some communication (not shown). In order to *directly* send the respective parts of each mapper's output to the responsible reducer process via channels, a unidirectional $m : n$ communication must be set up. Each process creates a list of m channels and passes them on to the caller. The latter thus receives a whole matrix of channels (one line received from each worker process) and passes them on to the workers column-wise. Intermediate data can now be partitioned as before, and intermediate grouped pairs directly sent to the worker responsible for the respective part.

Google's productive implementation realises this $m : n$ communication by shared files. The whole data subset processed by one mapper is pre-grouped into buckets, each for one reducer process, and written to a distributed mass storage system (GFS), to be fetched by reducer processes later. While this is clearly essential for fault tolerance (in order to restart computations without data being lost in failing machines), we consider accumulating all intermediate data on mass storage a certain disadvantage in performance and infrastructure requirements.

5 Measurements for Example Applications

Example applications of Google map-reduce can be taken from literature [15][16], which, however, tend to apply very simple reduce functions and can be realised using the elementary map-reduce without keys as well. We have chosen two example programs with non-trivial key-based reduction from literature, after comparing performance for a simple map-fold computation. We have tested:

- a simple map-fold computation (sum of Euler totient values),
- the NAS-EP benchmark (using key-based reduction),
- the K-Means implementation (using key-based reduction)

on a Beowulf cluster² with up to 32 Intel Pentium 4 SMP processor elements (PEs) running at 3 GHz with 512 MB RAM and a Fast Ethernet interconnection. Trace visualisations show activity profiles of the PEs (y-axis) over time (x-axis) in seconds.

Sum of Euler totient values. The `sumEuler` program is a straightforward map-fold computation, summing up values of the Euler function $\varphi(k)$. $\varphi(k)$ tells how many $j < k$ are relatively prime to k , and the test program

² At Heriot-Watt University Edinburgh.

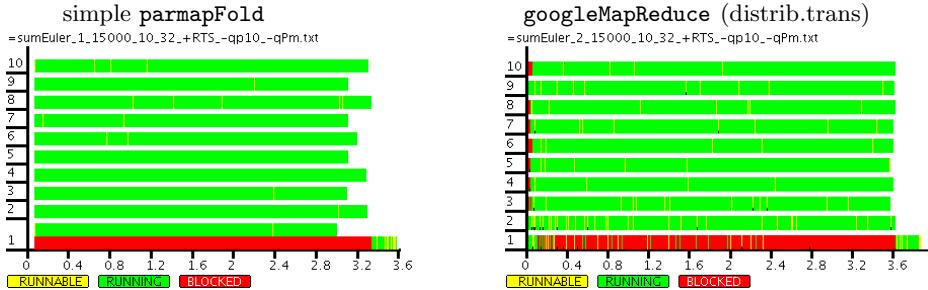


Fig. 5. Sum (reduction) of Euler totient (map) applications, input 15k

computes it naïvely instead of using prime factorisation. So, the program computes $\sum_{k=1}^n \varphi(k) = \sum_{k=1}^n |\{j < k \mid \text{gcd}(k, j) = 1\}|$, or in Haskell syntax:

```
result = foldl1 (+) (map phi [1..n])
phi k = length (filter (primeTo k) [1..(k-1)])
```

Fig. 5 shows runtime traces for two versions: the straightforward parallel map-fold skeleton and the Google map-reduce version with distributed transposition.

Both programs perform well on our measurement platform. The Google map-reduce implementation suffers from the overhead of distributing the map tasks (which is almost entirely eliminated in the map-fold version), whereas the other version obviously exposes uneven workload due to the static task distribution.

NAS-EP benchmark. NAS-EP (Embarrassingly Parallel) [17] is a transformation problem where two-dimensional statistics are accumulated from a large number of Gaussian pseudo-random numbers, and one of only few problems which profit from the per-key reduction functionality provided by the Google map-reduce skeleton (keys indicating the 10 different square annuli). Fig. 6 shows the results, using worker functions for reduced input data size, and the skeleton version with distributed transposition. Workload distribution is fair, and the system profits from Eden’s stream functionality to finish early in the reducer processes.

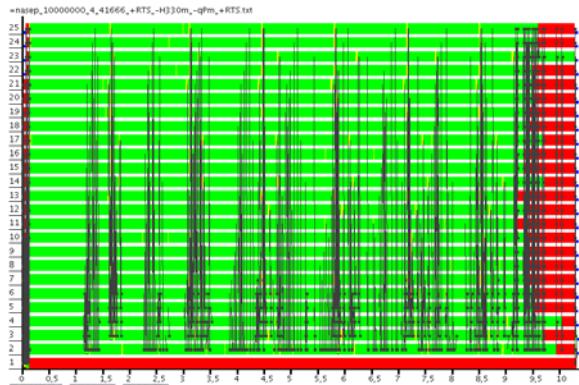


Fig. 6. NAS-EP benchmark, input 10⁸

Parallel k-means. The final example, *k*-means, illustrates that the additional reduction using the `COMBINE` function (as opposed to simply applying `REDUCE`

```

mAP :: Int -> Vector -> [(Int,Vector)]
mAP _ vec = [(1+minIndex (map (distance vec) centroids),vec)]

cOMBINE :: Int -> [Vector] -> Maybe (Int,Vector)
cOMBINE _ vs = Just (length vs, center vs)

rEDUCE :: Int -> [(Int,Vector)] -> Maybe (Int,Vector)
rEDUCE _ vs = Just (round w,v)
  where vs' = map (\(k,v) -> (fromIntegral k,v)) vs :: [(Double,Vector)]
        (w,v) = foldl1' combineWgt vs'

combineWgt :: (Double,Vector) -> (Double,Vector) -> (Double,Vector)
combineWgt (k1,v1) (k2,v2) = (k1+k2,zipWith (+) (wgt f v1) (wgt (1-f) v2))
  where {f = 1/(1+(k2/k1)); wgt x = map (*x) }

```

Fig. 7. Parameter functions for the parallel k -means algorithm

twice) is necessary, but might render algorithms more complicated. It also shows that Google map-reduce, however expressive, can turn out a suboptimal choice. The input of the k -means benchmark is a collection of data vectors (and arbitrary irrelevant keys). A set of k cluster centroids is chosen randomly in the first iteration. Parameterising the function to map with these centroids, the map part computes distances from the input vector to all centroids and yields the ID of the nearest centroid as the key, leaving the data as the value. The reduction, for each centroid, computes the mean vector of all data vectors assigned to the respective cluster to yield a set of k new cluster centroids, which is used in the next iteration, until the cluster centroids finally converge to desired precision.

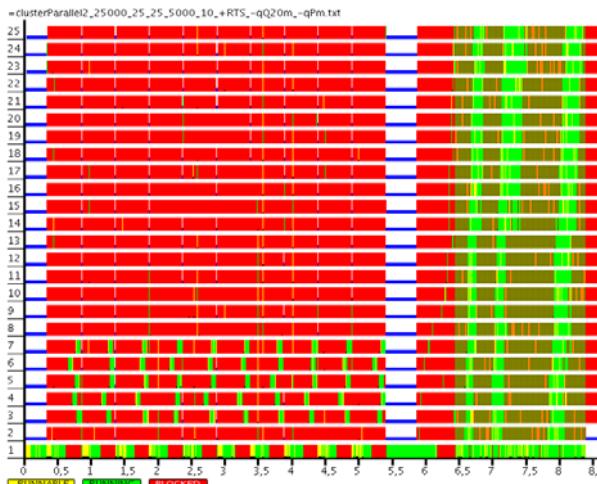


Fig. 9. k -means algorithm, Google map-reduce followed by iteration version

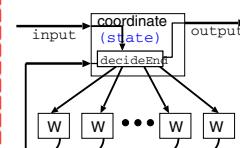


Fig. 8. Iteration skeleton

Fig. 7 shows the parameter functions for implementing k -means with Google map-reduce. Instead of computing sub-centroids by a simple sum, the vector subsets are precombined to a sub-centroid with added weight (cOMBINE function), to avoid numeric overflows and imprecision. The final reduction rEDUCE then uses the weight, which indicates the (arbitrary) number of vectors assigned to the sub-centroid, and combines centroids by a *weighted* average.

However, k -means is not suitable for the Google map-reduce skeleton on distributed memory machines. The problem is, the algorithm works iteratively in fixed steps, and amounts to setting up a new Google map-reduce instance for each iteration. These globally synchronised iteration steps and the skeleton setup overhead dominate the runtime behaviour, especially because the tasks are data vectors, sent and received again in each step. An iteration skeleton (as depicted in Fig. 8) should be used instead, with the advantage that the data vectors become initialisation data for the skeleton. The execution trace in Fig. 9 shows both versions for 25000 data vectors: first 10 iterations of the Google map-reduce version, then a version using an iteration skeleton (which performs around 90 iterations in shorter time). Communication of the data vectors completely eats up parallel speedup, whereas they are initialisation data in the second version.

6 Related Work

We have already cited the basic references for the skeleton in question throughout its description and in the introduction. Originally published in 2004 [6], the Google employees Dean and Ghemawat have updated their publication recently with ACM [7], providing more recent figures of the data throughput in the productive implementation. The Hadoop project [18] provides an open-source java implementation. A more thorough description of its functionality is provided by Ralf Lämmel [8], first published online in 2006.

Both the original description by the Google authors and Ralf Lämmel discuss inherent parallelism of the Google map-reduce skeleton. While Lämmel presents substantial work for a sound understanding and specification of the skeleton, his parallelisation ideas remain at a high level, at times over-simplified, and he does not discuss any concrete implementation. The original Google work restricts itself to describing and quantifying the existing parallelisation, but gives details about the physical setup, the middleware in use, and error recovery strategies.

Several publications have adopted and highlight Google map-reduce, with different focus. An evaluation in the context of machine-learning applications can be found in [15]. Ranger et al. [16] present an implementation framework for Google map-reduce for multicores, *Phoenix*. They report superlinear speedups (ranges up to 30) on a 24-core machine, achieved by adjusting the data sizes to ideal values for good cache behaviour. Other authors have recognised the advantages of the high-level programming model and propose it for other custom architectures: Cell [19] and FPGAs [20].

7 Conclusions

A comprehensive overview of Google map-reduce and its relation to algorithmic skeletons has been given. We have discussed two implementations for Google map-reduce, one following earlier work, and an optimised Eden version. As our runtime analyses for some example applications show, the skeleton implementation delivers good performance and is easily applicable to a range of problems. Implementations using explicitly parallel functional languages like Eden open the view on computation structure and synchronisation, which largely facilitates skeleton customisation and development.

The popularity of Google map-reduce nicely shows the ease and flexibility of skeletons to a broader audience; and has found good acceptance in mainstream development. On the other hand, the popularity of just one skeleton may sometimes mislead application development, not considering alternative skeletons. It turns out that the full generality of the Google map-reduce skeleton is often not needed, and other skeletons are more appropriate. Nevertheless, we consider Google map-reduce a big step for skeleton programming to finally get adequate attention, as a mathematically sound high-level programming model for novel parallel architectures.

Acknowledgements: We thank the anonymous referees, Phil Trinder, Kevin Hammond and Hans-Wolfgang Loidl, for helpful comments on earlier versions.

References

1. Cole, M.I.: Algorithmic Skeletons: Structured Management of Parallel Computation. Research Monographs in Parallel and Distributed Comp. MIT Press, Cambridge (1989)
2. Rabhi, F.A., Gorlatch, S. (eds.): Patterns and Skeletons for Parallel and Distributed Computing. Springer, Heidelberg (2003)
3. MPI-2: Extensions to the message-passing interface. Technical report, University of Tennessee, Knoxville (July 1997)
4. Gamma, E., Helm, R., Johnson, R.E., Vlissides, J.M.: Design patterns: Abstraction and reuse of object-oriented design. In: Nierstrasz, O. (ed.) ECOOP 1993. LNCS, vol. 707, pp. 406–431. Springer, Heidelberg (1993)
5. Mattson, T.G., Sanders, B.A., Massingill, B.L.: Patterns for Parallel Programming. SPS. Addison-Wesley, Reading (2005)
6. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. In: OSDI 2004, Sixth Symp. on Operating System Design and Implementation (2004)
7. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Communications of the ACM 51(1), 107–113 (2008)
8. Lämmel, R.: Google’s MapReduce programming model - Revisited. Science of Computer Programming 70(1), 1–30 (2008)
9. Poldner, M., Kuchen, H.: Scalable farms. In: Proceedings of ParCo 2005. NIC Series, vol. 33 (2005)
10. Benoit, A.: ESkel — The Edinburgh Skeleton Library. University of Edinburgh (2007), <http://homepages.inf.ed.ac.uk/abenoit1/eSkel/>

11. Michaelson, G., Scaife, N., Bristow, P., King, P.: Nested Algorithmic Skeletons from Higher Order Functions. *Parallel Algorithms and Appl.* 16, 181–206 (2001)
12. Blelloch, G.: Programming Parallel Algorithms. *Communications of the ACM* 39(3), 85–97 (1996)
13. Danelutto, M., DiCosmo, R., Leroy, X., Pelagatti, S.: Parallel functional programming with skeletons: the OCamlP3L experiment. In: ACM workshop on ML and its applications (1998)
14. Loogen, R., Ortega-Mallén, Y., Peña-Marí, R.: Parallel Functional Programming in Eden. *Journal of Functional Programming* 15(3), 431–475 (2005)
15. Chu, C.T., Kim, S.K., Lin, Y.A., Yu, Y., Bradski, G., Ng, A.Y., Olukotun, K.: Map-reduce for machine learning on multicore. In: Schölkopf, B., Platt, J., Hoffman, T. (eds.) *Advances in Neural Information Processing Systems*, vol. 19. MIT Press, Cambridge (2007)
16. Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., Kozyraki, C.: Evaluating mapreduce for multi-core and multiprocessor systems. In: *HPCA 2007: Int. Symp. on High Performance Computer Architecture*, pp. 13–24. IEEE Computer Society, Los Alamitos (2007)
17. Bayley, D., Barszcz, E., et al.: NAS Parallel Benchmarks. Technical Report RNR-94-007, NASA Advanced Supercomputing (NAS) Division (1994)
18. Apache Hadoop Project. Web page, <http://hadoop.apache.org>
19. de Kruijf, M., Sankaralingam, K.: MapReduce for the Cell B.E. Architecture. Technical Report TR1625, Computer Sc.Dept., Madison, WI (2007)
20. Yeung, J.H., Tsang, C., Tsui, K., Kwan, B.S., Cheung, C.C., Chan, A.P., Leong, P.H.: Map-reduce as a programming model for custom computing machines. In: *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE Computer Society, Los Alamitos (2008)

A Lower Bound for Oblivious Dimensional Routing

Andre Osterloh

BTC AG
Escherweg 5
26121 Oldenburg
Germany
osterloh@andre-osterloh.de

Abstract. In this work we consider deterministic oblivious dimensional routing algorithms on d -dimensional meshes. In oblivious dimensional routing algorithms the path of a packet depends only on the source and destination node of the packet. Furthermore packets use a shortest path with a minimal number of bends. We present an $\Omega(kn^{(d+1)/2})$ step lower bound for oblivious dimensional k - k routing algorithms on d -dimensional meshes for odd $d > 1$ and show that the bound is tight.

1 Introduction

Routing is one of the most studied topics in parallel computation. One of the best studied routing problems is the problem where each processor is source and destination of at most k packets, the k - k *routing problem*. Many different approaches to solve this routing problem have been studied [3]. In [17] the concept of oblivious routing was introduced. In this routing strategy the path of each packet is determined by its source and destination position. This property of oblivious routing algorithms is interesting, since it allows one to design simple and hence practical algorithms. Furthermore, it is of theoretical interest how fast routing problems under such restrictions can be solved. Hence oblivious routing was considered in several publications e.g. [12, 8, 11, 14, 15]. It was shown that the simplicity of oblivious routing has its costs in the running time. In [6] Kaklamanis, Krizanc, and Tsantilas have shown that every oblivious k - k routing algorithm on a network with N nodes and degree r needs at least $\Omega(k \cdot \sqrt{N}/r)$ steps.

Meshes are very attractive networks due to their simple and regular structure. In a d -dimensional $n \times \dots \times n$ mesh n^d processors are connected by a d -dimensional grid of bidirectional communication links. The degree of a d -dimensional mesh is $2d \in O(1)$ and hence for $d > 1$ an $\Omega(kn^{d/2})$ step lower bound for oblivious k - k routing on these networks exists. For $d = 1$ the bisection bound results in a $\Omega(kn)$ step lower bound. For the case $d = 1$ optimal $O(kn)$ step oblivious k - k routing algorithms that use $O(k)$ buffers are known [3]. For all $d > 1$ asymptotically optimal $O(kn^{d/2})$ step oblivious k - k routing algorithms that use $O(k)$ buffers per node exist [13, 10, 5].

An interesting question is how simple the paths can be chosen such that the $O(kn^{d/2})$ step bound can be achieved. In [4] *elementary-path* oblivious routing algorithms are introduced and an $\Omega(n^2)$ step lower bound for oblivious elementary-path 1-1 routing algorithms for the case $d = 3$ is shown. In an elementary path routing algorithm every packet uses a shortest path with a minimal number of bends on its way from source to destination. In [10] elementary-path routing algorithms are called dimensional routing algorithms and an asymptotically optimal $O(n)$ step oblivious routing algorithm for the case $k = 1$ and $d = 2$ is presented.

In this work we extend the lower bound of [4] to k - k routing and all odd $d > 1$, i.e., we give a tight lower bound of $\Omega(kn^{(d+1)/2})$ steps for oblivious dimensional k - k routing on d -dimensional $n \times \dots \times n$ meshes. The proof technique used to obtain the lower bound differs from that in [4].

This work is organized as follows. In Section 2, we define d -dimensional meshes, the model and oblivious dimensional routing algorithms. In Section 3, we present the new tight lower bound. Section 4 concludes.

2 Basic Definitions

Throughout this paper we model d -dimensional meshes by directed graphs. The nodes of the graph represent the processors and the edges represent communication links. For a directed graph $G = (V, E)$, and an edge $e = (x, y) \in E$ from x to y we write $src(e) = x$ and $dst(e) = y$. Here src stands for source and dst for destination. Furthermore, we use the notation src (dst) to denote the source (destination) node of packets. For $n \in \mathbf{N}$ we define $[n] = \{0, 1, \dots, n - 1\}$ and S_n as the symmetric group¹ of degree n .

2.1 Model of Computation

The processors operate in a synchronous fashion and communicate by sending packets over the unidirectional communication links. We assume that at most one packet can be transmitted by a link from source to destination in one step. In a single step, a processor receives a number of packets that are sent to it via a link in the previous step, performs some amount of internal computation, and sends a number of packets to neighbouring processors. Furthermore processors are able to store an unbounded number of packets in *buffers*.

2.2 Routing Problems

A routing problem on $G = (V, E)$ can be described by a triple (\mathcal{P}, src, dst) , where \mathcal{P} is a set of packets, and src, dst are mappings from \mathcal{P} to V . In a routing problem (\mathcal{P}, src, dst) each packet $p \in \mathcal{P}$ is loaded in node $src(p)$ initially and has to be sent to node $dst(p)$.

¹ S_n is the set of all permutations of $[n]$.

A k - k routing problem is a routing problem in which each node is source and destination of at most k packets, i.e., $|src^{-1}(\{v\})| \leq k$ and $|dst^{-1}(\{v\})| \leq k$ for all $v \in V$.

In the literature, a formal definition of oblivious routing algorithms is hard to find. Very often oblivious routing algorithms are described as algorithms where the path of a packet only depends on its source and destination and is independent of other packets, e.g. [7,14,6,10]. We give a formal definition of an oblivious routing algorithm. Before we can do so, we have to define paths.

Definition 1 (path). Let G be a directed graph and $l \in \mathbb{N}$. A path w in G is a (possibly empty) sequence $w = e_0e_1 \cdots e_{l-1}$ of edges of G such that $dst(e_i) = src(e_{i+1})$ for all $i \in [l-1]$.

If w is a non empty sequence we call w a path from $src(e_0)$ to $dst(e_{l-1})$. If w is an empty sequence we call w a path from x to x for all nodes x of G . \square

Definition 2 (oblivious routing algorithm). Let $G = (V, E)$ be a graph and $Path(G)$ be the set of all paths in G . Let \mathcal{A} be a deterministic k - k routing algorithm and let

$$path(\mathcal{A}, (\mathcal{P}, src, dst)) : \mathcal{P} \rightarrow Path(G)$$

be a mapping such that $path(\mathcal{A}, (\mathcal{P}, src, dst))(p)$ is the path on which a packet $p \in \mathcal{P}$ is sent from $src(p)$ to $dst(p)$ by \mathcal{A} when the algorithm solves the k - k routing problem (\mathcal{P}, src, dst) .

Algorithm \mathcal{A} is an oblivious k - k routing algorithm on G iff a mapping

$$\pi : V \times V \rightarrow Path(G)$$

exists, such that for all k - k routing problems (\mathcal{P}, src, dst) the following holds:

$$\forall p \in \mathcal{P} : \pi(src(p), dst(p)) = path(\mathcal{A}, (\mathcal{P}, src, dst))(p).$$

\square

2.3 Mesh-Connected Networks

One of the most studied networks is the two-dimensional $n \times n$ mesh. In this work we consider its natural generalizations, d dimensional meshes.

Definition 3 (d -dimensional mesh, edge of i -th dimension). The d -dimensional mesh of side length n , denoted by $M_{d,n}$, is a directed graph with node set $[n]^d$ and edge set

$$\{((x_0, \dots, x_{d-1}), (y_0, \dots, y_{d-1})) \mid \forall i \in [d] : x_i, y_i \in [n], \sum_{i=0}^{d-1} |x_i - y_i| = 1\}.$$

An edge $((x_0, \dots, x_{d-1}), (y_0, \dots, y_{d-1}))$ of $M_{d,n}$ where $|x_i - y_i| = 1$ is called an edge of the i -th dimension. \square

Since we use directed graphs a bidirectional communication link between two nodes x and y in a mesh is modelled by two directed edges (x, y) and (y, x) .

2.4 Dimensional Routing

In [4] oblivious *elementary-path* routing algorithms are introduced. In an elementary-path routing algorithm the packets use a minimal number of bends on their way from source to destination. Hence in an elementary path algorithm all packets are routed on a shortest path from source to destination. Furthermore, each elementary path in $M_{d,n}$ can be decomposed into at most d subpaths where each subpath consists of edges of (at most) one dimension. The decomposition into paths of different dimension motivates the notation *dimensional-path* algorithms [10] for elementary-path algorithms. In [10] elementary-path routing algorithms are called *weakly-dimensional* routing algorithms.

Before we give a formal definition of oblivious weakly-dimensional and *strongly-dimensional-path* routing algorithms we introduce π -paths for a permutation π . A π -path is a shortest path that can be decomposed into d subpaths p_i such that each p_i only consists of edges of dimension $\pi(i)$.

Definition 4 (π -path in $M_{d,n}$). Let $a, b \in [n]^d$, $\pi \in S_d$ and w be a shortest path from a to b in $M_{d,n}$. Path w is a π -path from a to b iff there exist (possibly empty) paths p_0, \dots, p_{d-1} in $M_{d,n}$ such that

- $w = p_0 p_1 \dots p_{d-1}$,
- $\forall i \in [d]:$ All edges of p_i are edges of the $\pi(i)$ -th dimension.

□

In Figure 1 we give examples of π -paths. Note that not every shortest path is a π -path for a permutation π . Furthermore, a shortest path could be a π -path for more than one permutation (see Figure 1).

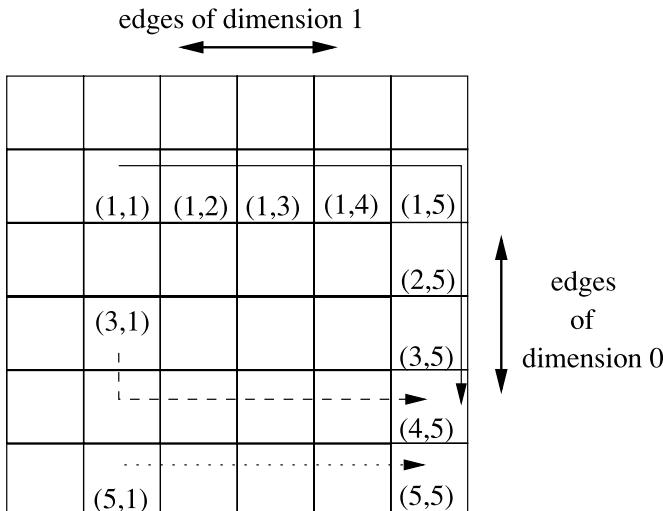


Fig. 1. Some π -paths in $M_{2,6}$. The solid line is a π_1 -path from $(1, 1)$ to $(4, 5)$ for $\pi_1(0) = 1$ and $\pi_1(1) = 0$. The dashed line is a π_2 -path from $(3, 1)$ to $(4, 5)$ for $\pi_2 = id$. Finally, the dotted line is a π_1 -path and a π_2 -path from $(5, 1)$ to $(5, 5)$.

Definition 5 (weakly-dimensional). An oblivious routing algorithm \mathcal{A} on $M_{d,n}$ is weakly-dimensional iff for all packets p there is a $\pi \in S_d$ such that \mathcal{A} routes p on a π -path from $\text{src}(p)$ to $\text{dst}(p)$. \square

If all packets in a weakly-dimensional routing algorithm use the same permutation π , we get a strongly-dimensional routing algorithm. The well known greedy routing algorithm [4] is an example for an oblivious strongly-dimensional routing algorithm.

Definition 6 (strongly-dimensional). An oblivious routing algorithm \mathcal{A} on $M_{d,n}$ is strongly-dimensional iff there is a $\pi \in S_d$ such that for all packets p algorithm \mathcal{A} routes p on a π -path from $\text{src}(p)$ to $\text{dst}(p)$. \square

3 Lower Bound

In [4][10][12] oblivious weakly-dimensional 1-1 routing algorithms are considered. In [4] it was shown that any oblivious elementary-path 1-1 routing algorithm on $M_{3,n}$ needs $\Omega(n^2)$ steps. We extend this result to k - k routing on $M_{d,n}$ for all odd d . We use a proof technique different from [4]. Without the restriction to elementary path the lower bound for oblivious routing on $M_{d,n}$ is $\Omega(kn^{d/2})$ [6].

Theorem 1. For odd d any oblivious and weakly-dimensional k - k routing algorithm on $M_{d,n}$ needs $\Omega(kn^{(d+1)/2})$ steps.

Proof. For $d = 1$ the result follows by the distance and bisection bound. We assume for the rest of the proof that d is odd and greater than one. Let \mathcal{A} be an oblivious and weakly-dimensional k - k routing algorithm on $M_{d,n}$.

For $t \in [n]^d, \pi \in S_d$, we define:

$$S(t, \pi) := \{s \in [n]^d \mid \text{Algorithm } \mathcal{A} \text{ sends a packet with source } s \text{ and destination } t \text{ on a } \pi\text{-path from } s \text{ to } t\}.$$

For every $t \in [n]^d$ there are n^d possible sources. There are $d!$ permutations $\pi \in S_d$. So

$$\forall t \in [n]^d. \exists \pi \in S_d. |S(t, \pi)| \geq \frac{n^d}{d!}. \quad (1)$$

Since (1) holds for all $t \in [n]^d$ there exists at least one π such that $|S(t, \pi)| \geq n^d/d!$ holds for $n^d/d!$ elements of $[n]^d$. So

$$\exists T \subseteq [n]^d, |T| \geq \frac{n^d}{d!}. \exists \pi \in S_d. \forall t \in T. |S(t, \pi)| \geq \frac{n^d}{d!}. \quad (2)$$

Without loss of generality we can assume in (2) that $\pi = \text{id}$. If $\pi \neq \text{id}$, we simply permute the dimensions according to π in the rest of the proof. Let $d = 2m + 1, m \geq 1$.

For $x \in [n]^m$ we define $M_x := \{(x, y_1, \dots, y_{m+1}) \in [n]^d \mid y_1, \dots, y_{m+1} \in [n]\}$ and $\overline{M}_x := \{(y_1, \dots, y_{m+1}, x) \in [n]^d \mid y_1, \dots, y_{m+1} \in [n]\}$. In $M_{d,n}$ each M_x and \overline{M}_x consists of n^{m+1} nodes. Furthermore there are n^m sets M_x and \overline{M}_x , each.

Now we take a T and π that fulfills (2). Since the family $(M_x)_{x \in [n]^m}$ partitions the nodes of $M_{d,n}$ at least one of the n^m sets M_x have a cut of size $(n^d/d!)/n^m = n^{m+1}/d!$ with T . So:

$$\begin{aligned} \exists x \in [n]^m. \exists T \subseteq [n]^d, |T| \geq \frac{n^d}{d!}. \exists \pi \in S_d. \\ (\forall t \in T. |S(t, \pi)| \geq \frac{n^d}{d!}) \wedge |T \cap M_x| \geq \frac{n^{m+1}}{d!}. \end{aligned} \quad (3)$$

We take such x , T and π from (3) and write T_x for $T \cap M_x$. Remember that we assume $\pi = id$. Since the family $(\overline{M}_y)_{y \in [n]^m}$ partitions the nodes of $M_{d,n}$ it follows $|S(t, id)| = \sum_{y \in [n]^m} |S(t, id) \cap \overline{M}_y|$ for all $t \in T_x$. From (3) we have $|S(t, id)| \geq n^d/d!$ for all $t \in T_x$. Now assume that a $t' \in T_x$ exists such that there are less than $n^m/(2d!)$ elements $y \in [n]^m$ such that $|S(t', id) \cap \overline{M}_y| \geq n^{m+1}/(2d!)$. For this $t' \in T_x$ it follows $n^d/d! \leq |S(t', id)| = \sum_{y \in [n]^m} |S(t', id) \cap \overline{M}_y| < \frac{n^m}{2d!} n^{m+1} + n^m \frac{n^{m+1}}{2d!} = n^d/d!$. So:

$$\exists x \in [n]^m. \forall t \in T_x. |\{y \in [n]^m : |S(t, id) \cap \overline{M}_y| \geq \frac{n^{m+1}}{2d!}\}| \geq \frac{n^m}{2d!}. \quad (4)$$

Again we take x , T and π from (3) and write T_x for $T \cap M_x$. For $y \in [n]^m$ we define $T_{x,y} := \{t \in T_x : |S(t, id) \cap \overline{M}_y| \geq \frac{n^{m+1}}{2d!}\}$. By (4) for all $t \in T_x$ there are at least $n^m/(2d!)$ elements $y \in [n]^m$ such that $|S(t, id) \cap \overline{M}_y| \geq n^{m+1}/(2d!)$. Furthermore, there are at least $n^{m+1}/d!$ elements in T_x and n^m possibilities for y . Since $\frac{\frac{n^m}{2d!} \cdot \frac{n^{m+1}}{d!}}{n^m} = \frac{n^{m+1}}{2(d!)^2}$ we get:

$$\exists x, y \in [n]^m. |T_{x,y}| \geq \frac{n^{m+1}}{2(d!)^2}. \quad (5)$$

Let (s_1, \dots, s_d) be the source, (t_1, \dots, t_d) be the target, and (a_1, \dots, a_d) be the actual position of a packet. After correcting the first m dimensions of a packet the following holds for the actual position: $a_1 = t_1, \dots, a_m = t_m, a_{m+1} = s_{m+1}, \dots, a_d = s_d$. While correcting dimension $m+1$ only a_{m+1} in the actual position of a packet changes. After correcting $m+1$ dimensions the following holds for the actual position: $a_1 = t_1, \dots, a_{m+1} = t_{m+1}, a_{m+2} = s_{m+2}, \dots, a_d = s_d$.

Now assume that a k - k problem and $x, y \in [n]^m$ exists that fulfills the following property (a):

- (a) After correcting the first m dimensions there are $k \frac{n^{m+1}}{2(d!)^2}$ packets with an actual position $a \in L := \{(x, i, y) \in [n]^d \mid i \in [n]\}$ and while correcting the dimension $m+1$ these packets have to travel a total distance of $\Omega(kn^{m+2})$.

There are at most n edges in L and at most one packet can go over an edge in one time step. So, for correcting the dimension $m+1$ of a k - k routing problem that fulfills (a), algorithm \mathcal{A} needs $\Omega(kn^{m+1})$ steps.

In the rest of the proof we show that a k - k routing problem exists that fulfills (a). We take x, y that fulfill (5). Consider a packet that has destination $t \in T_{x,y}$ and source in $S(t, id) \cap \overline{M}_y$. After the correction of the first m dimensions by algorithm \mathcal{A} the packet is on a node $(x, i, y) \in L$ for an $i \in [n]$ and has to

be transported to a node $(x, j, y) \in L$, $j \in [n]$. In the case that for a given destination $t = (x, y_1, \dots, y_{m+1}) \in T_{x,y}$, i.e. $j = y_1$, q sources in $\overline{M_y} \cap S(t, id)$ exist there exists at least one source such that a packet has to travel a distance of at least $\lfloor q/(2n^m) \rfloor$ on L , i. e. a source exists such that $|i - j| \geq \lfloor q/(2n^m) \rfloor$. Every $t \in T_{x,y}$ has at least $n^{m+1}/(2d!)$ sources in $S(t, id) \cap \overline{M_y}$ (see (4)). Two elements in $T_{x,y}$ could have the same source, so the number of sources we can choose from is reduced with every choice. So the q in (6) is $n^{m+1}/(2d!) - i$. Since $|T_{x,y}| \geq n^{m+1}/(2(d!)^2)$ we have at least $n^{m+1}/(2(d!)^2)$ source/destination pairs and the sum in (6) goes from 0 to $n^{m+1}/(2(d!)^2) - 1$. To every found source/destination pair we assign k packets. So we are able to construct a k - k problem such that the total distance the packets have to travel on L is at least

$$k \sum_{i=0}^{\frac{n^{m+1}}{2(d!)^2} - 1} \left\lfloor \frac{\frac{n^{m+1}}{2d!} - i}{2n^m} \right\rfloor \in \Omega(kn^{m+2}). \quad (6)$$

Hence we have proven (a). \square

Every oblivious and strongly-dimensional k - k routing algorithm is also an oblivious and weakly-dimensional k - k routing algorithm. Hence

Corollary 1. *For odd d any oblivious and strongly-dimensional k - k routing algorithm on $M_{d,n}$ needs $\Omega(kn^{\frac{d+1}{2}})$ steps.*

The routing algorithm in which all packets use an id -path is a strongly-dimensional oblivious routing algorithm. If we use the furthest destination first priority scheme this algorithm solves a k - k routing problem on $M_{d,n}$ in $O(kn^{\frac{d+1}{2}})$ steps for odd d and in $O(kn^{\frac{d}{2}})$ steps for even d . This can be seen by using the routing lemma of [16] since in our model we allow the processors to store an unlimited amount of packets in buffers. So the lower bound is tight.

4 Conclusion

In this work we presented a tight $\Omega(kn^{(d+1)/2})$ step lower bound for oblivious dimensional k - k routing on d -dimensional meshes for odd $d > 1$. Dimensional $\Omega(kn^{(d+1)/2})$ step oblivious k - k routing algorithms using an unbounded number of buffers exists. For $d > 2$ it is unknown whether the existing lower bounds for dimensional routing can be achieved with algorithms using $O(k)$ buffers.

References

1. Borodin, A., Hopcroft, J.E.: Routing, merging, and sorting on parallel models of computation. In: Proceedings of the 14th Annual ACM Symposium on the Theory of Computing, pp. 338–344 (1982)
2. Borodin, A., Raghavan, P., Schieber, B., Upfal, E.: How much can hardware help routing? In: In: Proceedings of the 25th Annual ACM Symposium on the Theory of Computing, pp. 573–582 (1993)

3. Grammatikakis, M.D., Hsu, D.F., Sibeyn, J.F.: Packet routing in fixed-connection networks: A survey. *Journal of Parallel and Distributed Computing* 54(2), 77–132 (1998)
4. Iwama, K., Miyano, E.: A lower bound for elementary oblivious routing on three-dimensional meshes. *Journal of Algorithms* 39, 145–161 (2001)
5. Iwama, K., Miyano, E.: An $O(\sqrt{N})$ oblivious routing algorithm for 2-D meshes of constant queue-size. *Journal of Algorithms* 41, 262–279 (2001)
6. Kaklamanis, C., Krizanc, D., Tsantilas, T.: Tight bounds for oblivious routing in the hypercube. *Mathematical Systems Theory* 24(4), 223–232 (1991)
7. Krizanc, D.: Oblivious routing with limited buffer capacity. *Journal of Computer and System Sciences* 43(2), 317–327 (1991)
8. Krizanc, D., Peleg, D., Upfal, E.: A time-randomness tradeoff for oblivious routing. In: *Proceedings of the 20th Annual ACM Symposium on the Theory of Computing*, pp. 93–102 (1988)
9. Leighton, F.T.: *Introduction to Parallel Algorithms and Architectures*. Morgan Kaufman, San Mateo (1992)
10. Litman, A., Moran-Schein, S.: Fast, minimal, and oblivious routing algorithms on the mesh with bounded queues. *Journal of Interconnection Networks* 2(4), 445–469 (2001)
11. Meyer auf der Heide, F., Scheideler, C.: Communication in parallel systems. In: Král, J., Bartosek, M., Jeffery, K. (eds.) *SOFSEM 1996. LNCS*, vol. 1175, pp. 16–34. Springer, Heidelberg (1996)
12. Osterloh, A.: Oblivious routing on d -dimensional meshes. In: *Proceedings of the 8th International Colloquium on Structural Information and Communication Complexity*, pp. 305–320. Carleton Scientific (2001)
13. Osterloh, A.: Aspects of k - k routing in Meshes and OTIS Networks. Ph.d. thesis, Technical University of Ilmenau (2002)
14. Parberry, I.: An optimal time bound for oblivious routing. *Algorithmica* 5, 243–250 (1990)
15. Rajasekaran, S., Tsantilas, T.: Optimal routing algorithms for mesh-connected processor arrays. *Algorithmica* 8, 21–38 (1992)
16. Sibeyn, J.F., Kaufmann, M.: Deterministic 1- k routing on meshes. In: Enjalbert, P., Mayr, E.W., Wagner, K.W. (eds.) *STACS 1994. LNCS*, vol. 775, pp. 237–248. Springer, Heidelberg (1994)
17. Valiant, L.G.: A scheme for fast parallel communication. *SIAM Journal on Computing* 11(2), 350–361 (1982)

Topic 13

High-Performance Networks

Introduction

Cees de Laat*, Chris Develder*, Admela Jukan*, and Joe Mambretti*

This topic is devoted to communication issues in scalable compute and storage systems, such as parallel computers, networks of workstations, and clusters. All aspects of communication in modern systems were solicited, including advances in the design, implementation, and evaluation of interconnection networks, network interfaces, system and storage area networks, on-chip interconnects, communication protocols, routing and communication algorithms, and communication aspects of parallel and distributed algorithms. In total 15 papers were submitted to this topic of which we selected the 7 strongest papers. We grouped the papers in two sessions of 3 papers each and one paper was selected for the best paper session. We noted a number of papers dealing with changing topologies, stability and forwarding convergence in source routing based cluster interconnect network architectures. We grouped these for the first session. The authors of the paper titled: “Implementing a Change Assimilation Mechanism for Source Routing Interconnects” propose a mechanism that can obtain the new topology, and compute and distribute a new set of fabric paths to the source routed network end points to minimize the impact on the forwarding service. The article entitled “Dependability Analysis of a Fault-tolerant Network Reconfiguration Strategy” reports on a case study analyzing the effects of network size, mean time to node failure, mean time to node repair, mean time to network repair and coverage of the failure when using a 2D mesh network with a fault-tolerant mechanism (similar to the one used in the BlueGene/L system), that is able to remove rows and/or columns in the presence of failures. The last paper in this session: “RecTOR: A New and Efficient Method for Dynamic Network Reconfiguration” presents a new dynamic reconfiguration method, that ensures deadlock-freedom during the reconfiguration without causing performance degradation such as increased latency or decreased throughput. The second session groups 3 papers presenting methods, protocols and architectures that enhance capacities in the Networks. The paper titled: “NIC-assisted Cache-Efficient Receive Stack for Message Passing over Ethernet” presents the addition of multiqueue support in the Open-MX receive stack so that all incoming packets for the same process are treated on the same core. It then introduces the idea of binding the target end process near its dedicated receive queue. In general this multiqueue receive stack performs better than the original single queue stack, especially on large communication patterns where multiple processes are involved and manual binding is difficult. The authors of: “A Multipath Fault-Tolerant Routing Method for High-Speed

* Topic Chairs.

Interconnection Networks” focus on the problem of fault tolerance for high-speed interconnection networks by designing a fault tolerant routing method. The goal was to solve a certain number of link and node failures, considering its impact, and occurrence probability. Their experiments show that their method allows applications to successfully finalize their execution in the presence of several faults, with an average performance value of 97% with respect to the fault-free scenarios. The paper: “Hardware implementation study of the Self-Clocked Fair Queuing Credit Aware (SCFQ-CA) and Deficit Round Robin Credit Aware (DRR-CA) scheduling algorithms” proposes specific implementations of the two schedulers taking into account the characteristics of current high-performance networks. A comparison is presented on the complexity of these two algorithms in terms of silicon area and computation delay. Finally we selected one paper for the special paper session: “A Case Study of Communication Optimizations on 3D Mesh Interconnects”. In this paper the authors present topology aware mapping as a technique to optimize communication on 3-dimensional mesh interconnects and hence improve performance. Results are presented for OpenAtom on up to 16,384 processors of Blue Gene/L, 8,192 processors of Blue Gene/P and 2,048 processors of Cray XT3.

A Case Study of Communication Optimizations on 3D Mesh Interconnects

Abhinav Bhatelé, Eric Bohm, and Laxmikant V. Kalé

Department of Computer Science
University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
`{bhatele, ebohm, kale}@illinois.edu`

Abstract. Optimal network performance is critical to efficient parallel scaling for communication-bound applications on large machines. With wormhole routing, **no-load** latencies do not increase significantly with number of hops traveled. Yet, we, and others have recently shown that in presence of **contention**, message latencies can grow substantially large. Hence task mapping strategies should take the topology of the machine into account on large machines. In this paper, we present topology aware mapping as a technique to optimize communication on 3-dimensional mesh interconnects and hence improve performance.

Our methodology is facilitated by the idea of object-based decomposition used in Charm++ which separates the processes of decomposition from mapping of computation to processors and allows a more flexible mapping based on communication patterns between objects. Exploiting this and the topology of the allocated job partition, we present mapping strategies for a production code, OpenAtom to improve overall performance and scaling. OpenAtom presents complex communication scenarios of interaction involving multiple groups of objects and makes the mapping task a challenge. Results are presented for OpenAtom on up to 16,384 processors of Blue Gene/L, 8,192 processors of Blue Gene/P and 2,048 processors of Cray XT3.

1 Introduction

A significant number of the largest supercomputers in use today, including IBM's Blue Gene family and Cray's XT family, employ a 3D mesh or torus topology. With tens of thousands of nodes, messages may have to travel many tens of hops before arriving at their destinations. With the advances in communication technology, especially wormhole routing, it was observed that the *latency* of communication was almost unaffected by the number of hops traveled by a message [12]. However, the fraction of *bandwidth* occupied by the message is proportional to the number of hops (links) traversed. Increased contention for bandwidth might result in longer latencies.

Consider a computation in which each processor sends one message to some other processor, in a data permutation pattern (so each processor also receives one message). For a thousand nodes organized in a $10 \times 10 \times 10$ 3D torus, the

Table 1. Execution time per step of OPENATOM on Blue Gene/L (CO mode) without topology aware mapping (System: WATER_32M_70Ry)

Cores	512	1024	2048	4096	8192
Time (secs)	0.274	0.189	0.219	0.167	0.129

average number of hops traveled by a random message (i.e. the average inter-node distance) is 7.5. In a torus, a message needs to travel at most 5 hops in each dimension, leading to an average of 2.5 per dimension. So, if we organize our computation so that each message travels only one hop, we will use 7.5 times smaller global bandwidth than in the case of a random communication pattern. This was not as significant an issue when the number of nodes was relatively small and the processors were slower¹. But for today's large machines with faster processors, the issue becomes much more significant: on a machine with 64,000 nodes organized in a $40 \times 40 \times 40$ 3D torus, the average inter-node distance is 30, and that is the ratio of bandwidth used by the two communication patterns. Further, with faster processors, the need for delivered bandwidth is higher. As the communication starts to occupy a large fraction of the available bandwidth, the contention in the network increases and message delivery gets delayed [3].

In this context, it is important to map computation to the processors to not just minimize the overall communication volume, but also the average number of hops traveled by the bytes communicated. Even though the utility of doing this may be apparent to programmers, the significance of the impact is probably more than most programmers expect. Our methodology builds upon object-based decomposition used in CHARM++ [4] and related programming models, including Adaptive MPI (AMPI) [5]. This separates the processes of decomposition from mapping of computation to processors and allows a more flexible mapping based on communication patterns between objects.

In this paper, we first present the abstraction of object-based decomposition in CHARM++ and an API which provides a uniform interface for obtaining topology information at runtime on four different machines – Cray XT3, Cray XT4, IBM Blue Gene/L and IBM Blue Gene/P. This API can be used by user-level codes for task mapping and is independent of the programming model being used (MPI, OpenMP, CHARM++ or something else). We then demonstrate topology aware mapping techniques for a communication intensive application: OPENATOM, a production Car-Parrinello *ab initio* Molecular Dynamics (CPAIMD) code used by scientists to study properties of materials and nano-scale molecular structures for biomimetic engineering, molecular electronics, surface catalysis, and many other areas [6][7][8][9].

Initial scaling studies of OPENATOM on Blue Gene/L uncovered inadequate parallel efficiency for a small system with 32 atoms (Table II). Further analysis isolated the cause to poor communication performance, which naturally led us

¹ This fact had made early work on topology aware mapping obsolete. But with large machines, bandwidth effects have again become important.

to consider the network topology. We consider 3D torus topologies in this paper but not irregular networks or flat topologies. For logarithmic topologies (such as fat trees), the need to pay attention to topology may be smaller because the maximum number of hops between nodes tends to be small. Also, there is no support for user level derivation of topology for most fat-tree networks so any implementation would be specific to an individual cluster layout.

2 Previous Work

There has been considerable research on the task mapping problem. It has been proven that the problem is NP-complete and computationally equivalent to the general graph embedding problem. Heuristic techniques like pairwise exchange were developed in the 80s by Bokhari [10] and Aggarwal [11]. These schemes, however, are not scalable when mapping millions of objects to thousands of processors. This problem has been handled by others using recursive partitioning [12] and graph contraction [13] by mapping sub-graphs to a subset of processors. Physical optimization techniques like simulated annealing [14] and genetic algorithms [15] are very effective but can take very long to arrive at optimal results. Results in the 80s and 90s were not demonstrated on real machines and even when they were, they were targeted towards small sized machines. They also did not consider real applications. With the emergence of large parallel machines, we need to revisit these techniques and build upon them, on a foundation of real machines, in the context of real applications.

A lot of work until the 90s was focused on hypercube networks [11, 14]. The development of large parallel machines like Blue Gene/L, XT3, XT4 and Blue Gene/P has led to the re-emergence of mapping issues. Both application and system developers have evolved mapping techniques for Blue Gene/L [16, 17, 18, 19, 20]. Yu [20] and Smith [19] discuss embedding techniques for graphs onto the 3D torus of BG/L which can be used by the MPI Topology functions. Weisser et al. [21] present an analysis of topology aware job placement techniques for XT3. However, our work is one of the first for task mapping on XT3. This work builds on our previous work [22] demonstrating the effectiveness of topology aware task mapping for a 3D Stencil kernel. It presents a case study for OPENATOM, a production quantum chemistry code, and demonstrates high returns using topology aware schemes. Our earlier publication on OPENATOM [23] demonstrates the effectiveness of such schemes on BG/L. In this paper, we present results for XT3, BG/L and BG/P for multiple systems including a non-benchmark simulation.

On machines like Blue Gene/L and Blue Gene/P, obtaining topology information is simple and an interface is available to the programmers. The API described in this paper provides a wrapper for these and additional functionality as we shall see in Section 3.1. However, on Cray XT machines, there is no interface for topology information, in accordance with the widespread, albeit mistaken idea, that topology mapping is not important on fast Cray machines. For XT machines, our API uses lower level system calls to obtain information about allocated partitions at runtime. To the best of our knowledge, there is no

published work describing such functionality for the Cray machines. We believe that this information will be useful to programmers running on Cray machines. Also, the API provides a uniform interface which works on all these machines which hides architecture specific details from the application programmer. This API can be used as a library for CHARM++, MPI or any other parallel program.

3 Charm++ Arrays: A Useful Abstraction for Mapping

Parallelizing an application consists of two tasks: 1. decomposition of the problem into a large number of sub-problems to facilitate efficient parallelization to thousands of processors, 2. mapping of these sub-problems to physical processors to ensure load balance and minimum communication. Object-based decomposition separates the two tasks and gives independent control over both of them. In this paper, we use the CHARM++ runtime which allows the application developer to decompose the problem into objects and the CHARM++ runtime does a default mapping of objects to processors.

The basic unit of computation in CHARM++ is called a *chare* (simply referred to as an “object” in this paper) which can be invoked through remote function calls. The application developer decomposes the problem into chares or objects and the CHARM++ runtime does a default mapping of objects to processors. Each processor can have multiple objects which facilitates overlap of computation and communication. This default mapping does not have any information about the topology of the machine. The user can override the default mapping with more intelligent schemes that take the topology of the machine into account.

3.1 Topology Manager API: Runtime Information

Mapping of communication graphs onto the processor graph requires information about the machine topology at runtime. The application should be able to query the runtime to get information like the dimensions of the allocated processor partition, mapping of ranks to physical nodes etc. However, the mapping interface should be simple and should hide machine-specific details from the application. The Topology Manager API in CHARM++ provides a uniform interface to the application developer and hence the application just knows that the job partition is a 3D torus or mesh topology. Application specific task mapping decisions require no architecture or machine specific knowledge (BG/L or XT3 for example).

The Topology Manager API in CHARM++ provides different functions which can be grouped into the following categories:

- 1. Size and properties of the allocated partition:** At runtime, the application needs to know the dimensions of the allocated partition (`getDimNX`, `getDimNY`, `getDimNZ`), number of cores per node (`getDimNT`) and whether we have a torus or mesh in each dimension (`isTorusX`, `isTorusY`, `isTorusZ`).
- 2. Properties of an individual node:** The interface also provides calls to convert from ranks to physical coordinates and vice-versa (`rankToCoordinates`, `coordinatesToRank`).

3. Additional Functionality: Mapping algorithms often need to calculate number of hops between two ranks or pick the closest rank to a given rank from a list. Hence, the API provides functions like `getHopsBetweenRanks`, `pickClosestRank` and `sortRanksByHops` to facilitate mapping algorithms.

We now discuss the process of extracting this information from the system at runtime and why is it useful to use the Topology Manager API on different machines:

IBM Blue Gene machines: On Blue Gene/L and Blue Gene/P [24], topology information is available through system calls to the “BGLPersonality” and “BGPPersonality” data structures, respectively. It is useful to use the Topology Manager API instead of the system calls for two reasons. First, these system calls can be expensive (especially on Blue Gene/L) and so it is advisable to avoid doing too many of them. The API does a few system calls to obtain enough information so that it can construct the topology information itself. It is useful to use the API instead of expensive system calls throughout the execution.

Cray XT machines: Cray machines have been designed with a significant overall bandwidth, and possibly for this reason, documentation for topology information was not readily available at the installations we used. We hope that the information provided here will be useful to other application programmers.

Obtaining topology information on XT machines is a two step process: 1. Getting the node ID (`nid`) corresponding to a given MPI rank (`pid`) which tells us which physical node a given MPI rank is on. This can be done through different system calls on XT3 and XT4: `cnos_get_nidpid_map` available through “catamount/cnos_mpi_os.h” and `PMI_Portals_get_nidpid_map` available from “pmi.h”. These calls provide a map for all ranks in the current job and their corresponding node IDs. 2. The second step is obtaining the physical coordinates for a given node ID. This can be done by using the system call `rca_get_meshcoord` from “rca_lib.h”. Once we have information about the physical coordinates for all ranks in the job, the API derives information such as the extent of the allocated partition by itself (this assumes that the machine has been reserved and we have a contiguous partition).

The API provides a uniform interface which works on all the above mentioned machines which hides architecture specific details from the application programmer. This API can be used as a library for CHARM++, MPI or any other parallel program. The next section describes the use of object-based decomposition and the Topology Manager API in a production code.

4 OpenAtom: A Case Study

An accurate understanding of phenomena occurring at the quantum scale can be achieved by considering a model representing the electronic structure of the

atoms involved. The CPAIMD method [25] is one such algorithm which has been widely used to study systems containing $10-10^3$ atoms. To achieve a fine-grained parallelization of CPAIMD, computation in OPENATOM [23] is divided into a large number of objects, enabling scaling to tens of thousands of processors. We will look at the parallel implementation of OPENATOM, explain the communication involved and then analyze the benefit from topology aware mapping of its objects.

In an *ab initio* approach, the system is driven by electrostatic interactions between the nuclei and electrons. Calculating the electrostatic energy involves computing several terms. Hence, CPAIMD computations involve a large number of phases with high inter-processor communication: (1) quantum mechanical kinetic energy of non-interacting electrons, (2) Coulomb interaction between electrons or the Hartree energy, (3) correction of the Hartree energy to account for the quantum nature of the electrons or the exchange-correlation energy, and (4) interaction of electrons with atoms in the system or the external energy. These phases are discretized into a large number of objects which generate a lot of communication, but ensures efficient interleaving of work. The entire computation is divided into ten phases which are parallelized by decomposing the physical system into fifteen *chare arrays*. For a detailed description of this algorithm please refer to [23].

4.1 Communication Dependencies

The ten phases referred to in the previous section are parallelized by decomposing the physical system into fifteen *chare arrays* of different dimensions (ranging between one and four). A simplified description of five of these arrays (those most relevant to the mapping) follows:

1. **GSpace and RealSpace:** These represent the g-space and real-space representations of each of the electronic states [25]. Each electronic state is represented by a 3D array of “complex numbers”. OPENATOM decomposes this data into a 2D *chare array* of objects. Each object holds a plane of one of the states (see Figure 1). The *chare arrays* are represented by $G(s, p)$ [$n_s \times N_g$] and $R(s, p)$ [$n_s \times N$] respectively. GSpace and RealSpace interact through

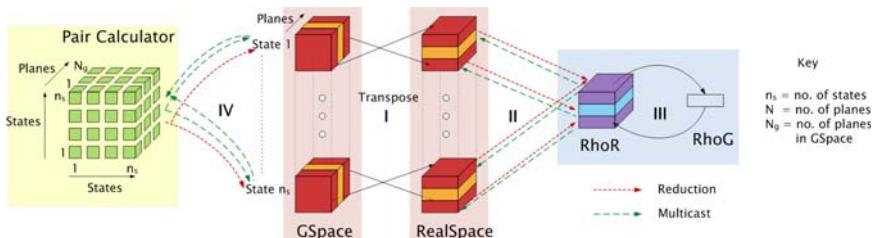


Fig. 1. Decomposition of the physical system into chare arrays (only important ones shown for simplicity) in OPENATOM

transpose operations (as part of a Fast Fourier Transform) in Phase I and hence all planes of one state of GSpace interact with all planes of the same state of RealSpace.

2. **RhoG and RhoR:** They are the g-space and real-space representations of electron density and are decomposed into 1D and 2D *chare arrays* respectively. They are represented as $G_\rho(p)$ and $R_\rho(p, p')$. RealSpace interacts with RhoR through reductions in Phase II. RhoG is obtained from RhoR in Phase III through two transposes.
3. **PairCalculators:** These 3D *chare arrays* are used in phase IV. They communicate with GSpace through multicasts and reductions. They are represented as $P_c(s, s', p)$ [$n_s \times n_s \times N_g$]. All elements of the GSpace array with a given state index interact with all elements of the PairCalculator array with the same state in one of their first two dimensions.

4.2 Mapping

OPENATOM provides us with a scenario where the load on each object is static (under the CPAIMD method) and the communication is regular and clearly understood. Hence, it should be possible to intelligently map the arrays in this application to minimize inter-processor communication and maintain load balance. OPENATOM has a default mapping scheme, but it should be noted that the default mapping is far from random. It is the mapping scheme used on standard fat-tree networks, wherein objects which communicate frequently are co-located on processors within the constraints of even distribution. This reduces the total communication volume. It only lacks a model for considering the relative distance between processors in its mapping considerations. We can do better than the default mapping by using the communication and topology information at runtime. We now describe how a complex interplay (of communication dependencies) between five of the *chare arrays* is handled by our mapping scheme.

GSpace and RealSpace are 2D *chare arrays* with states in one dimension and planes in the other. These arrays interact with each other through transpose operations where all planes of one state in GSpace, $G(s, *)$ talk to all planes of the same state, $R(s, *)$ in RealSpace (state-wise communication). The number of planes in GSpace is different from that in RealSpace. GSpace also interacts with the PairCalculator arrays. Each plane of GSpace, $G(*, p)$ interacts with the corresponding plane, $P(*, *, p)$ of the PairCalculators (plane-wise communication) through multicasts and reductions. So, GSpace interacts state-wise with RealSpace and plane-wise with PairCalculators. If all planes of GSpace are placed together, then the transpose operation is favored, but if all states of GSpace are placed together, the multicasts/reductions are favored. To strike a balance between the two extremes, a hybrid map is built, where a subset of planes and states of these three arrays are placed on one processor.

Mapping GSpace and RealSpace Arrays: Initially, the GSpace array is placed on the torus and other objects are mapped relative to GSpace's mapping. The 3D torus is divided into rectangular boxes (which will be referred to as

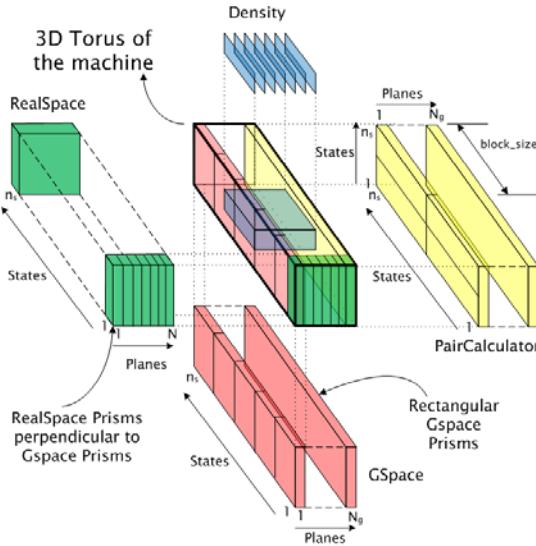


Fig. 2. Mapping of different arrays to the 3D torus of the machine

“prisms”) such that the number of prisms is equal to the number of the planes in GSpace. The longest dimension of the prism is chosen to be same as one dimension of the torus. Each prism is used for all states of one plane of GSpace. Within each prism for a specific plane, the states in $G(*, p)$ are laid out in increasing order along the long axis of the prism. Figure 2 shows the GSpace prisms (box at the bottom) being mapped along the long dimension of the torus (box in the center). Once GSpace is mapped, the RealSpace objects are placed. Prisms perpendicular to the GSpace prisms are created which are formed by including processors holding all planes for a particular state of GSpace, $G(s, *)$. These prisms (box on the left) are perpendicular to the GSpace prisms and the corresponding states of RealSpace, $R(s, *)$ are mapped on to these prisms.

Mapping of Density Arrays: RhoR objects communicate with RealSpace plane-wise and hence $R_\rho(p, *)$ have to be placed close to $R(*, p)$. To achieve this, we start with the centroid of the prism used by $R(*, p)$ and place RhoR objects in proximity to it. RhoG objects, $G_\rho(p)$ are mapped near RhoR objects, $R_\rho(p, *)$ but not on the same processors as RhoR to maximize overlap. The density computation is inherently smaller and hence occupies the center of the torus (box on the top in Figure 2).

Mapping PairCalculator Arrays: Since PairCalculator and GSpace objects interact plane-wise, the aim is to place $G(*, p)$ and $P(*, *, p)$ nearby. *Chares* with indices $P(s1, s2, p)$ are placed around the centroid of $G(s1, p), \dots, G(s1 + block_size, p)$ and $G(s2, p), \dots, G(s2 + block_size, p)$. This minimizes the hop-count for the multicast and reduction operations. The result of this

mapping co-locates each plane of PairCalculators (box on the right in Figure 2) with its corresponding plane of GSpace objects within the GSpace prisms.

The mapping schemes discussed above substantially reduce the hop-count for different phases. They also restrict different communication patterns to specific prisms within the torus, thereby reducing contention and ensuring balanced communication throughout the torus. State-wise and plane-wise communication is confined to different (orthogonal) prisms. This helps avoid scaling bottlenecks as we will see in Section 4.3. These maps perform no better (and generally slightly worse) than the default maps on architectures which have more uniform network performance, such as Ethernet or Infiniband.

Time Complexity: Although maps are created only once during application start-up, they must still be efficient in terms of their space and time requirements. The memory cost of these maps grows linearly (3 integers per object) with the number of objects, which is a few megabytes in the largest system studied. The runtime cost of creating the most complex of these maps is $O(pn^2 \log(n))$ where n is the number of objects and p the number of processors. Despite this complexity, this time is sufficiently small that generating the maps for even the largest systems requires only a few minutes. As an optimization, once created, maps can be stored and reloaded in subsequent runs to minimize restart time. Offline creation of maps using even more sophisticated techniques and adapting these ideas to other topologies is an area of future work.

4.3 Comparative Analysis of OpenAtom

To analyze the effects of topology aware mapping in a production science code we studied the strong scaling (fixed problem size) performance of OPENATOM with and without topology aware mapping. Two benchmarks commonly used in the CPMD community: the minimization of WATER_32M_70Ry and WATER_256M_70Ry were used. The benchmarks simulate the electronic structure of 32 molecules and 256 molecules of water, respectively, with a standard g-space spherical cutoff radius of $|\mathbf{g}|_{cut}^2 = 70$ Rydberg (Ry) on the states. To illustrate

Table 2. Execution time per step (in secs) of OPENATOM on Blue Gene/L (CO mode)

Cores	WATER_32M_70Ry		WATER_256M_70Ry		GST_BIG		
	Default	Topology	Default	Topology	Default	Topology	
512	0.274	0.259	-	-	-	-	-
1024	0.189	0.150	19.10	16.4	10.12	8.83	
2048	0.219	0.112	13.88	8.14	7.14	6.18	
4096	0.167	0.082	9.13	4.83	5.38	3.35	
8192	0.129	0.063	4.83	2.75	3.13	1.89	
16384	-	-	3.40	1.71	1.82	1.20	

Table 3. Execution time per step (in secs) of OPENATOM on Blue Gene/P (VN mode)

Cores	WATER_32M_70Ry		WATER_256M_70Ry	
	Default	Topology	Default	Topology
256	0.395	0.324	-	-
512	0.248	0.205	-	-
1024	0.188	0.127	10.78	6.70
2048	0.129	0.095	6.85	3.77
4096	0.114	0.067	4.21	2.17
8192	-	-	3.52	1.77

Table 4. Performance (time per step in secs) of OPENATOM on XT3

Cores	WATER_32M_70Ry			WATER_256M_70Ry			GST_BIG		
	Default	Topology	Default	Topology	Default	Topology	Topology		
Single core per node									
512	0.124	0.123	5.90	-	5.37	4.82	3.86		
1024	0.095	0.078	4.08	-	3.24	2.49	2.02		
Two cores per node									
256	0.226	0.196	-	-	-	-	-		
512	0.179	0.161	7.50	-	6.58	6.28	5.06		
1024	0.144	0.114	5.70	-	4.14	3.51	2.76		
2048	0.135	0.095	3.94	-	2.43	2.90	2.31		

that the performance improvements extend beyond benchmarks to production science systems, we also present results for GST_BIG, which is a system being studied by our collaborator, Dr Glenn J. Martyna. GST_BIG consists of 64 molecules of Germanium, 128 molecules of Antimony and 256 molecules of Tellurium at cutoff radius of $|\mathbf{g}|_{cut}^2 = 20$ Ry on the states.

Blue Gene/L (IBM T. J. Watson) runs are done in co-processor (CO) mode to use a single core per node. Single core per node runs were chosen to highlight interconnect performance and to facilitate fair comparisons between the two machines. Blue Gene/P (Intrepid at ANL) runs were done in VN mode using all four cores per node. Cray XT3 (BigBen at PSC) runs are done in two modes: single core per node (SN) and two cores per node (VN).

As shown in Table 2, performance improvements from topology aware mapping for Blue Gene/L (BG/L) can be quite significant. As the number of cores and likewise, the diameter of the torus grows, the performance impact increases until it is a factor of two faster for WATER_32M_70Ry at 2048 and for WATER_256M_70Ry at 16384 cores. There is a maximum improvement of 40% for GST_BIG. The effect is not as strong in GST_BIG due to the fact that the time step in this system is dominated by a subset of the orthonormalization process which has not been optimized extensively, but even a 40% improvement represents a substantial improvement in time to solution.

Performance improvements on Blue Gene/P (Table 3) are similar to those observed on BG/L. The improvement for WATER_256M_70Ry is not as remarkable as on BG/L but for WATER_256M_70Ry, we see a factor of 2 improvement starting at 2048 cores. The absolute numbers on BG/P are much better than on BG/L partially because of the increase in processor speeds but more due to the better interconnect (higher bandwidth and DMA engine). The performance for WATER_256M_70Ry at 1024 cores is 2.5 times better on BG/P than on BG/L. This is when comparing the VN mode on BG/P to the CO mode on BG/L. If we use only one core per node on BG/P, the performance difference is even greater, but the higher core per node count, combined with the DMA engine and faster network make single core per node use less interesting on BG/P.

The improvements from topology awareness on Cray XT3, presented in Table 4 are comparable to those on BG/L and BG/P. The improvement of 20% and 18.8% on XT3 for WATER_256_70Ry and GST_BIG at 1024 cores is greater than the improvement of 14% and 12% respectively on BG/L at 1024 cores in spite of a much faster interconnect.

The improvement trends plotted in Figure 3 lead us to project that topology aware mapping should yield improvements proportional to torus size on larger Cray XT installations. The difference in processor speeds is approximately a factor of 4 (XT3 2.6 Ghz, BG/L 700 Mhz), which is reflected in the performance for the larger grained OPENATOM results on XT3 when comparing single core per node performance. The difference in network performance is approximately a factor of 7 (XT3 1.1 GB/sec, BG/L 150 MB/sec), when considering delivered bandwidth as measured by HPC Challenge [26] ping pong. This significant difference in absolute speed and computation/bandwidth ratios does not shield the XT3 from performance penalties from topology ignorant placement schemes.

As discussed in prior sections, OPENATOM is highly communication bound. Although CHARM++ facilitates the exploitation of the available overlap and latency tolerance across phases, the amount of latency tolerance inevitably drops as the computation grain size is decreased by the finer decomposition required for larger parallel runs. It is important to consider the reasons for these performance improvements in more detail. Figure 4(a) compares idle time as captured by

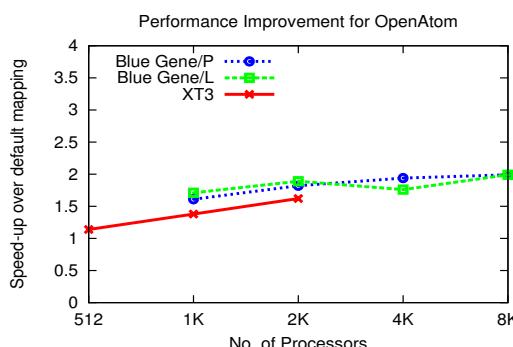


Fig. 3. Comparison of benefit by topology aware mapping (for WATER_256M_70Ry)

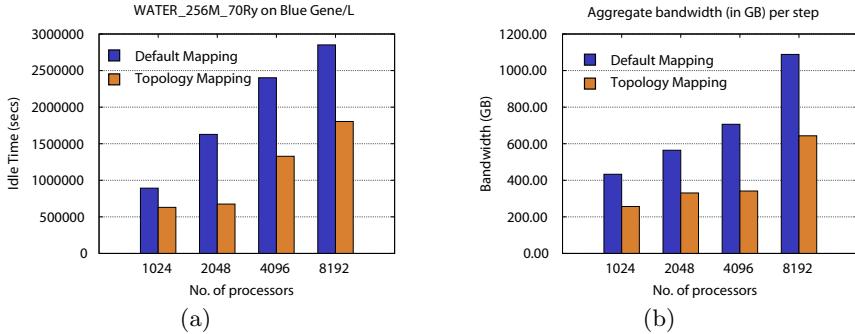


Fig. 4. (a) Effect of topology aware mapping on idle time and latency (OPENATOM running WATER_256M_70Ry on Blue Gene/L), (b) Effect of topology aware mapping on bandwidth (OPENATOM running WATER_256M_70Ry on Blue Gene/P)

the Projections profiling system in CHARM++ for OPENATOM on BG/L for the default mapping, versus the topology aware mapping. A processor is idle whenever it is waiting for messages to arrive. It is clear from Figure 4(a) that the factor of two speed increase from topology awareness is reflected directly in relative idle time and that the maximum speed increase which can be obtained from topology aware mapping is a reduction in the existing idle time.

It is illuminating to study the exact cause for this reduction in idle time. To that end, we ported IBM's High Performance Monitor library [27] for Blue Gene/P's Universal Performance Counters to Charm++, and enabled performance counters for a single time step in WATER_256M_70Ry in both topology aware and non-topology aware runs. We added the per node torus counters (BGP_TORUS_*_32BCHUNKS), to produce the aggregate bandwidth consumed by one step across all nodes to obtain the results in Figure 4(b). It is clear from the figure, that topology aware mapping results in a significant reduction, by up to a factor of two, in the total bandwidth consumed by the application. This more efficient use of the network is directly responsible for the reduction in latency due to contention and decreased idle time.

5 Conclusion and Future Work

In this paper we demonstrated that topology aware mapping can substantially improve performance for communication intensive applications on 3D torus networks. Significant improvements were shown for the OPENATOM code and the effectiveness of topology aware mapping was shown for both IBM Blue Gene and Cray XT architectures. Mapping was facilitated by the object-based virtualization in CHARM++ and the availability of the Topology Manager API.

OPENATOM has complex but relatively regular or structured communication. We think that it may be possible to develop general methodologies that deal with such structured communication. Unstructured static communication patterns, as represented by unstructured-mesh computations might need somewhat different

mapping techniques. Work in the future will involve developing an automatic mapping framework which can work in tandem with the topology manager interface. This would also require a study of a more diverse set of applications with different communication patterns. Further study will be given to characterizing network resource usage patterns with respect to those which are most affected by topology aware task mapping.

Acknowledgments

This work was supported in part by a DOE Grant B341494 funded by Center for Simulation of Advanced Rockets, DOE grant DE-FG05-08OR23332 through ORNL LCF, and a NSF Grant ITR 0121357 for Quantum Chemistry. This research was supported in part by NSF through TeraGrid [28] resources provided by NCSA and PSC through grants ASC050040N and MCA93S028. We thank Shawn T. Brown and Chad Vizino from PSC for help with system reservations and runs on BigBen. We also thank Fred Mintzer, Glenn Martyna and Sameer Kumar from IBM for access and assistance in running on the Watson Blue Gene/L. We also used running time on the Blue Gene/P at Argonne National Laboratory, which is supported by DOE under contract DE-AC02-06CH11357.

References

1. Greenberg, R.I., Oh, H.C.: Universal wormhole routing. *IEEE Transactions on Parallel and Distributed Systems* 08(3), 254–262 (1997)
2. Ni, L.M., McKinley, P.K.: A survey of wormhole routing techniques in direct networks. *Computer* 26(2), 62–76 (1993)
3. Bhatele, A., Kale, L.V.: An Evaluation of the Effect of Interconnect Topologies on Message Latencies in Large Supercomputers. In: Proceedings of Workshop on Large-Scale Parallel Processing (IPDPS 2009) (May 2009)
4. Kalé, L., Krishnan, S.: CHARM++: A Portable Concurrent Object Oriented System Based on C++. In: Paepcke, A. (ed.) *Proceedings of OOPSLA 1993*, September 1993, pp. 91–108. ACM Press, New York (1993)
5. Bhandarkar, M., Kale, L.V., de Sturler, E., Hoeflinger, J.: Object-Based Adaptive Load Balancing for MPI Programs. In: Alexandrov, V.N., Dongarra, J., Juliano, B.A., Renner, R.S., Tan, C.J.K. (eds.) *ICCS-ComputSci 2001*. LNCS, vol. 2074, pp. 108–117. Springer, Heidelberg (2001)
6. Pasquarello, A., Hybertsen, M.S., Car, R.: Interface structure between silicon and its oxide by first-principles molecular dynamics. *Nature* 396, 58 (1998)
7. De Santis, L., Carloni, P.: Serine proteases: An ab initio molecular dynamics study. *Proteins* 37, 611 (1999)
8. Saitta, A.M., Soper, P.D., Wasserman, E., Klein, M.L.: Influence of a knot on the strength of a polymer strand. *Nature* 399, 46 (1999)
9. Rothlisberger, U., Carloni, P., Doclo, K., Parrinello, M.: A comparative study of galactose oxidase and active site analogs based on QM/MM Car Parrinello simulations. *J. Biol. Inorg. Chem.* 5, 236 (2000)
10. Bokhari, S.H.: On the mapping problem. *IEEE Trans. Computers* 30(3), 207–214 (1981)

11. Lee, S.Y., Aggarwal, J.K.: A mapping strategy for parallel processing. *IEEE Trans. Computers* 36(4), 433–442 (1987)
12. Ercal, F., Ramanujam, J., Sadayappan, P.: Task allocation onto a hypercube by recursive mincut bipartitioning. In: *Proceedings of the 3rd conference on Hypercube concurrent computers and applications*, pp. 210–221. ACM Press, New York (1988)
13. Berman, F., Snyder, L.: On mapping parallel algorithms into parallel architectures. *Journal of Parallel and Distributed Computing* 4(5), 439–458 (1987)
14. Bollinger, S.W., Midkiff, S.F.: Processor and link assignment in multicomputers using simulated annealing. In: *ICPP* (1), pp. 1–7 (1988)
15. Arunkumar, S., Chockalingam, T.: Randomized heuristics for the mapping problem. *International Journal of High Speed Computing (IJHSC)* 4(4), 289–300 (1992)
16. Bhanot, G., Gara, A., Heidelberger, P., Lawless, E., Sexton, J.C., Walkup, R.: Optimizing task layout on the Blue Gene/L supercomputer. *IBM Journal of Research and Development* 49(2/3), 489–500 (2005)
17. Gygi, F., Draeger, E.W., Schulz, M., Supinski, B.R.D., Gunnels, J.A., Austel, V., Sexton, J.C., Franchetti, F., Kral, S., Ueberhuber, C., Lorenz, J.: Large-Scale Electronic Structure Calculations of High-Z Metals on the Blue Gene/L Platform. In: *Proceedings of the International Conference in Supercomputing*. ACM Press, New York (2006)
18. Bhatelé, A., Kalé, L.V., Kumar, S.: Dynamic Topology Aware Load Balancing Algorithms for Molecular Dynamics Applications. In: *23rd ACM International Conference on Supercomputing* (2009)
19. Smith, B.E., Bode, B.: Performance Effects of Node Mappings on the IBM Blue Gene/L Machine. In: Cunha, J.C., Medeiros, P.D. (eds.) *Euro-Par 2005*. LNCS, vol. 3648, pp. 1005–1013. Springer, Heidelberg (2005)
20. Yu, H., Chung, I.H., Moreira, J.: Topology mapping for Blue Gene/L supercomputer. In: *SC 2006: Proceedings of the ACM/IEEE conference on Supercomputing*, p. 116. ACM, New York (2006)
21. Weisser, D., Nystrom, N., Vizino, C., Brown, S.T., Urbanic, J.: Optimizing Job Placement on the Cray XT3. In: *48th Cray User Group Proceedings* (2006)
22. Bhatelé, A., Kalé, L.V.: Benefits of Topology Aware Mapping for Mesh Interconnects. *Parallel Processing Letters (Special issue on Large-Scale Parallel Processing)* 18(4), 549–566 (2008)
23. Bohm, E., Bhatelé, A., Kale, L.V., Tuckerman, M.E., Kumar, S., Gunnels, J.A., Martyna, G.J.: Fine Grained Parallelization of the Car-Parrinello ab initio MD Method on Blue Gene/L. *IBM Journal of Research and Development: Applications of Massively Parallel Systems* 52(1/2), 159–174 (2008)
24. IBM Blue Gene Team: Overview of the IBM Blue Gene/P project. *IBM Journal of Research and Development* 52(1/2) (2008)
25. Tuckerman, M.E.: Ab initio molecular dynamics: Basic concepts, current trends and novel applications. *J. Phys. Condensed Matter* 14, R1297 (2002)
26. Dongarra, J., Luszczek, P.: Introduction to the HPC Challenge Benchmark Suite. Technical Report UT-CS-05-544, University of Tennessee, Dept. of Computer Science (2005)
27. Salapura, V., Ganesan, K., Gara, A., Gschwind, M., Sexton, J., Walkup, R.: Next-Generation Performance Counters: Towards Monitoring Over Thousand Concurrent Events. In: *IEEE International Symposium on Performance Analysis of Systems and Software*, April 2008, pp. 139–146 (2008)
28. Catlett, C., et al.: TeraGrid: Analysis of Organization, System Architecture, and Middleware Enabling New Types of Applications. In: Grandinetti, L. (ed.) *HPC and Grids in Action*. IOS Press, Amsterdam (2007)

Implementing a Change Assimilation Mechanism for Source Routing Interconnects*

Antonio Robles-Gómez, Aurelio Bermúdez, and Rafael Casado

Instituto de Investigación en Informática de Albacete,

Universidad de Castilla-La Mancha, 02071 Albacete

{arobles, abermu, rcasado}@dsi.uclm.es

Abstract. Current high-performance interconnects are switch-based systems using either distributed or source routing. Usually, they incorporate a management mechanism in charge of detecting and assimilating any topological change. In order to restore the connectivity among endpoints, this mechanism must obtain the new topology, and compute and distribute a new set of fabric paths. It is desirable that the change assimilation process is simple, fast, deadlock-free and, above all, it must minimize the impact of the change on the network service. In this work we implement and evaluate a complete management mechanism for source routing networks that accomplishes all these requirements. The proposed scheme results from the combination of previous proposals for each task in the change assimilation process.

Keywords: High-performance networks, topology discovery, routing protocols, dynamic reconfiguration, performance evaluation.

1 Introduction

The communication subsystem in modern high-performance computing systems plays a vital and central role. Usually, this subsystem is based on a point-to-point switch-based technology. In recent years, some different technologies have emerged, based on distributed routing –InfiniBand (IBA) [1]– or source routing –Myrinet 2000 [2], Advanced Switching Interconnect (ASI) [3,4]–. From the point of view of hardware devices, the application of source routing reduces the complexity of switches, increasing the speed of the routing decision. In case of distributed routing, routes are more flexible and their management is easier.

In these environments, after the occurrence of a topological change (such as the failure or aggregation of a component), a new routing function, which is appropriate to the resulting topology, must be obtained and uploaded in the routing elements. In case of source routing, this consists in updating the routing

* This work has been jointly supported by the Spanish MEC and European Commission FEDER funds under grants “Consolider Ingenio-2010 CSD2006-00046” and “TIN2006-15516-C04-02”, and a FPI grant under “TIC2003-08154-C06-02”; by JCCM under grants “PREG-07-25”, “PII1C09-0101-9476” and “PII2I09-0067-3628”.

tables at fabric endpoints. The change assimilation process is typically performed by a centralized management entity, which is called *mapper* in Myrinet, *subnet manager* in InfiniBand, and *fabric manager* in ASI.

It is well known that a non optimized implementation for the tasks involved in the assimilation process could lead to a significant degradation in network performance. In previous works, we have developed separately efficient techniques for discovering the network topology, computing a new set of fabric paths, and reconfiguring the routing tables, considering the use of source routing networks. In this paper, we show that all these proposals are compatible, and can be combined in a complete management solution. The resulting mechanism is simple, fast, and keeps the fabric up and running after the occurrence of a topological change. The consequence is that, as we show in the performance evaluation section, the combined mechanism minimizes the impact that any change has on the network performance.

The remainder of this paper is organized as follows. Section 2 describes the strategies implemented for each phase involved in the proposed management mechanism. Next, Section 3 presents a detailed performance evaluation for this mechanism. Finally, the conclusions and future work are given in Section 4.

2 The Fabric Management Mechanism

The change assimilation process entails the topology discovery, path computation, and network reconfiguration tasks. Fig. 1 shows the flow chart of the complete process.

2.1 Change Detection and Topology Discovery

When a fabric device detects a change in the status of a local port, it may notify this event to the manager by means of a specific management packet. This is the detection technique that we have considered in this work. Another option could be that the manager polls periodically the status of fabric devices, looking for topological changes, similarly to the *sweeping* mechanism included in InfiniBand [1].

After that, the manager must obtain information about all active devices—endpoints and switches—physically connected to the fabric. To handle this task, a serialized discovery algorithm [5] was proposed for source routing environments. This algorithm explores the fabric devices sequentially, that is, the manager reads all the relevant configuration information from a single device before proceeding to the next device. As a consequence, there is only a management packet in the fabric at a given moment.

In [6], we proposed and analyzed an alternative parallel implementation for the topology discovery task, which significantly improves the serialized algorithm. In this case, the manager injects a new management packet as soon as an active port is discovered during the process, in order to obtain information about the device connected at the other end of the link. In this way, the discovery process spreads through several paths in parallel, exploring different devices simultaneously.

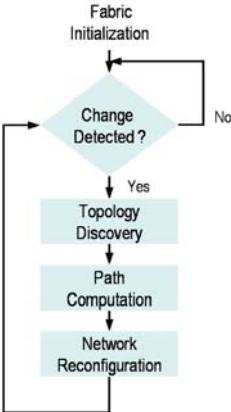


Fig. 1. Phases of a Fabric Management Mechanism

2.2 Path Computation

Once the manager updates its topological information, it must establish the set of paths that the endpoints will use to communicate among them. We assume the use of *up*/down** [7], a popular deadlock-free routing algorithm valid for any regular or irregular topology. It is based on an assignment of directions to the operational links in the fabric, which produces an acyclic directed graph with a single sink node, referred to as *correct graph* [8]. To avoid deadlocks, *up*/down** establishes that legal routes never use a link in the *up* direction after having used one in the *down* direction.

To compute the set of *up*/down** routes, in this work we have considered the *Fully Explicit Routing* (FER) algorithm [9], previously proposed for IBA networks and, recently, adapted to source routing environments. FER is a Dijkstra-based *up*/down** implementation which works as follows. For each destination node in the fabric, it considers the set of valid paths which can be used to reach that destination, starting from the rest of nodes. This is done by performing a conventional controlled flooding, and preventing those hops that involve a forbidden *down-up* transition.

Traditionally, the path computation task has been performed in a centralized way. In [10] we have proposed to execute this process in a distributed manner. This means that each fabric endpoint is in charge of obtaining its own set of paths. To enable distributed route computation, endpoints need to know the topological information that was gathered by the manager during the discovery process. For this reason, the manager must send this information to the fabric endpoints, instead of waiting for it to compute all the fabric paths and to distribute them. As a result, the task of computing the new routing function is shared among all endpoints and, therefore, topological changes are assimilated much faster.

In the literature, alternative proposals to compute a deadlock-free set of paths are described. For instance, the *Layered Shortest Path* (LASH) algorithm [11].

To achieve deadlock freedom, this technique separates the traffic in several layers by using different sets of virtual channels. Inside each layer, a subset of sources uses deadlock-free minimal routes to reach a subset of destinations. Recently, it was presented its adaptation to source routing environments [12].

2.3 Network Reconfiguration

The last phase involved in the change assimilation process consists in replacing the old routing function by the new one. In the literature, this process is traditionally referred to as *network reconfiguration*. It is well known that, although both functions are by themselves deadlock-free, updating fabric paths in an uncontrolled way may lead to deadlock situations since packets that belong to one of the routing functions may take turns that are not allowed in the another function [8].

Early reconfiguration mechanisms (designed for Autonet [7] and Myrinet [13] networks) solved this problem by emptying the network of packets before replacing the routing function. That is, packets that are routed according to the old function and packets that are routed according to the new function are not simultaneously present in the network. Such a simple approach is referred to as *static reconfiguration*, and has a very negative impact on the network service availability since for a period of time the network cannot accept packets.

During the last years, several schemes have been proposed for distributed routing systems in order to increase network availability during the change over from one routing function to another. These mechanisms allow injection of data traffic while the routing function is being updated, and are known as *dynamic reconfiguration* techniques. Updating the routing function while the network remains up and running, on the other hand, requires more advanced reconfiguration schemes in order to guarantee deadlock-freedom.

Next, we depict the most relevant dynamic reconfiguration techniques. The *Partial Progressive Reconfiguration* (PPR) [8] and *Skyline* [14] approaches aim to repair an uncorrected up*/down* graph which includes several sink nodes. The adaptation of these techniques to networks that use source routing is not trivial. In addition, *NetRec* [15] and *LORE* [16] have been specially designed for rerouting messages around a faulty node. Their implementation in source routing networks is not feasible. *Double Scheme* [17] requires several virtual channels to separate packets routed according to the old routing function from packets routed according to the new one. On the other hand, *Overlapping Reconfiguration* [18] needs to introduce a special packet (called a *token*) that governs the transition from the old routing function to the new one. This algorithm does not require distributed routing, but uses a distributed mechanism to update the VC selection. Recently, *Epoch-Based Reconfiguration* (EBR) [19] was also proposed for regressive deadlock recoveries.

On the other hand, we proposed the first dynamic reconfiguration technique [20] suitable for source routing networks. This proposal was named *Close Graph-based Reconfiguration* (CGR). In this strategy, we assume that the manager first computes a correct up*/down* graph starting from the previous one which has lost its

correctness because of the change. The resulting graph, referred to as *close graph*, has the property that each cycle in both graphs is broken in the same or a neighboring node. After that, the task of computing new paths (that can be executed in either a centralized or a distributed way) is performed by taking into account the restrictions imposed by both the old and new graphs. Finally, endpoint routing tables can be asynchronously updated without producing deadlocks. In [20] we formally prove that it is always possible to obtain the close graph and a routing function that satisfies the routing restrictions imposed by both graphs.

3 Performance Evaluation

In this section, we compare the performance of two fabric management mechanisms designed for source routing interconnects. The ‘initial’ mechanism performs a serialized exploration of the topology, computes the new routing tables in a centralized way, and finally updates them to the fabric endpoints by using static reconfiguration [21]. On the other side, the ‘proposed’ mechanism incorporates all the optimizations described in the previous section. In particular, after detecting a topological change, it performs a parallel discovery, obtains the routing tables in a distributed way, and updates them dynamically (using the CGR technique).

The performance evaluation has been carried out by using simulation techniques. Our interest here focuses on evaluating the cost of the change assimilation process, in terms of time and, above all, its impact over application traffic. Before presenting the simulation results, the simulation methodology is detailed.

3.1 Simulation Methodology

The simulation model [22] used for this work was developed with the OPNET Modeler [23], and embodies the physical and link layers of Advanced Switching, a recent example of source routing technology. The model implements 16-port multiplexed virtual cut-through switches, where each port uses a 1x lane (2.5 Gbps), and endpoints are connected to the fabric through a single port.

This model provides the necessary support to design fabric management mechanisms as defined in the Advanced Switching specification [3]. In particular, it includes management entities (fabric manager and device managers), device capabilities, and management packets. In addition, the model considers the time required by the management entities to process incoming management packets and perform tasks associated with the reception of such packets.

We have evaluated several regular fabric topologies, including meshes, tori, and fixed-arity fat-trees. A complete list is presented in Table 1, and Fig. 2 shows some examples. For meshes and tori, we assume that each border switch has an endpoint attached. Although the management mechanisms analyzed do not require the use of several virtual channels (VC), we have used four VCs per fabric port. One of them is dedicated to management traffic, and the remainder to data traffic [3]. Management traffic gets higher scheduling priority than data

Table 1. Topologies

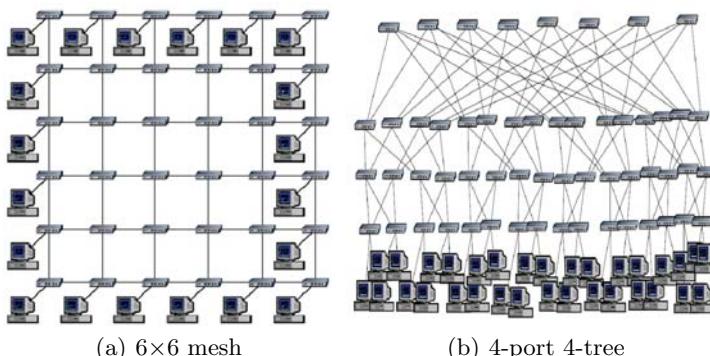
Topology	Switches	Endpoints	Total
3x3 mesh, 3x3 torus	9	8	17
4x4 mesh, 4x4 torus	16	12	28
6x6 mesh, 6x6 torus	36	20	56
8x8 mesh, 8x8 torus	64	28	92
9x9 torus	81	32	113
4-port 2-tree	6	8	14
4-port 3-tree	20	16	36
4-port 4-tree	56	32	88
8-port 2-tree	12	32	44

traffic in switches and therefore it will always take a similar time to assimilate a topological change, independently of the amount of data packets in the network. The size of the input and output buffers associated with each VC is 8 Kbytes.

For all simulations, the data packet length is fixed at 512 bytes. The traffic load applied is dependent on the fabric topology and the number (and size) of available VCs, representing 50% of the saturation rate in each case. Anyway, these values do not affect the relative behavior of the algorithms when they are comparatively evaluated. In addition, the packet generation rate was uniform, and traffic sources also used a uniform distribution to obtain the packet destination among all the active endpoints. Again, results were very similar when other traffic patterns are applied.

For each simulation there is an initial transient period in which fabric devices are activated and the fabric manager gathers the original topology. Later, a topological change (either the activation or deactivation of a randomly chosen fabric switch) is scheduled at time 2.0 seconds (after the fabric starts up). At this time, we have checked that for all cases the network has reached a steady state.

In order to increase the accuracy of the results, each experiment has been repeated several times for each fabric topology. The amount of simulation runs

**Fig. 2.** Example of two fabric topologies in OPNET

for each topology is the 50% of the number of physical nodes (both switches and endpoints) for both switch activations and deactivations, and averages have been drawn from the solution set and presented graphically.

3.2 Simulation Results

Fig. 3 shows the time required by both the initial and proposed management mechanisms, for all topologies evaluated in this work (Table II), to completely assimilate a topological change consisting in a switch activation (Fig. 3a) and deactivation (Fig. 3b), as a function of the fabric size. Change detection time has not been included in these results. The plots confirm that the proposed mechanism considerably reduces this statistic. The main reason for this reduction is the distributed path computation technique.

Fig. 4 shows the amount of data packets discarded by both the initial and proposed mechanisms, as a function of the fabric size. This parameter gives an indication of the level of service that a network can provide to applications after the occurrence of a topology change. As we can observe, when the proposed mechanism is applied (Fig. 4b and Fig. 4d), the number of data packets discarded is significantly lower. This is mainly due to the use of a dynamic reconfiguration technique. Furthermore, in case of a switch activation, the network does not discard any data packet.

In particular, the bars in this figure represent the quantities that relate to three different reasons for packets being discarded. They are labeled as ‘Protected Ports’, ‘Inactive Ports’ and ‘Inactive Tables’. The ‘Protected Ports’ label quantifies packets that are discarded when they reach (logically) inactive ports (i.e., in the *DL_Protected* state, in which ports can only receive and transmit management packets [3]). The ‘Inactive Ports’ label refers to packets that attempt to cross the switch that has been removed from the network, in order

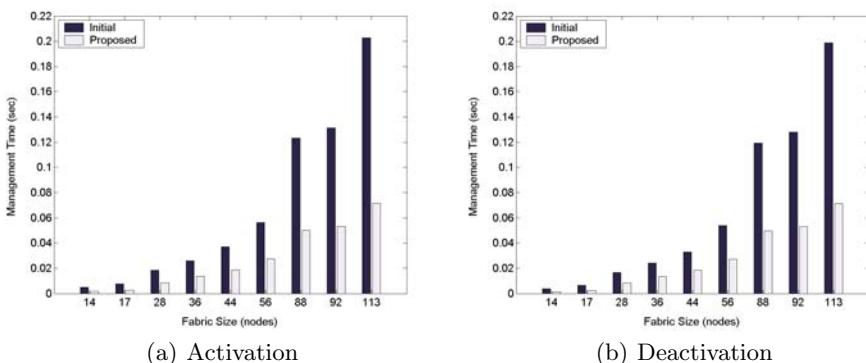


Fig. 3. Average time required by the evaluated management mechanisms to completely assimilate a topological change

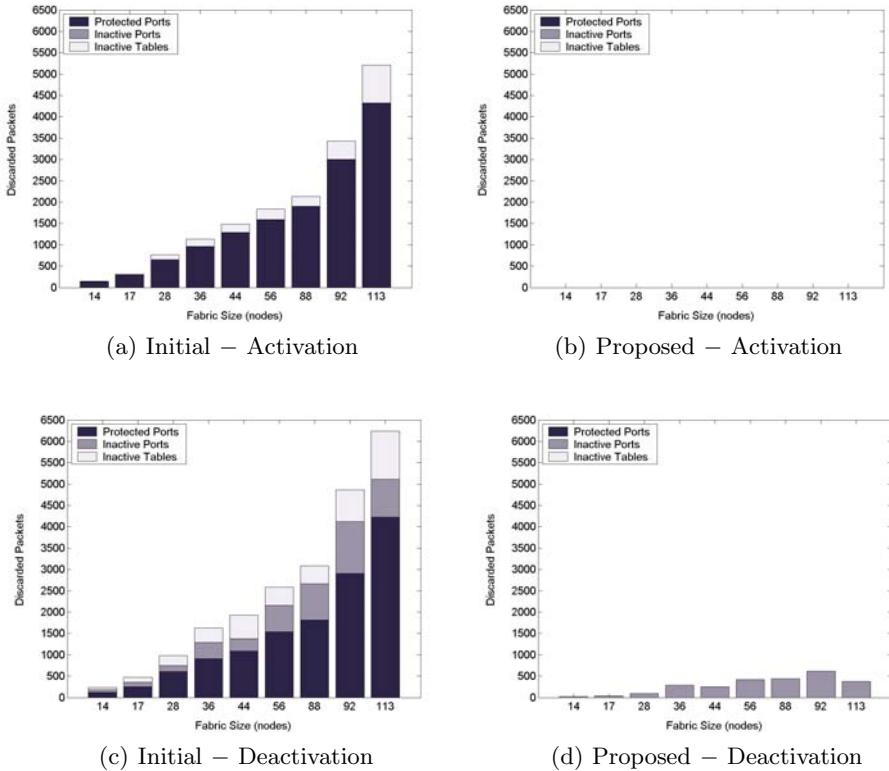
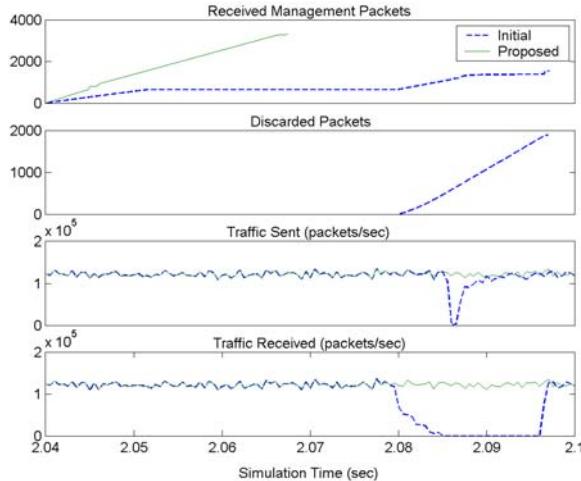


Fig. 4. Average amount of data packets discarded by the evaluated management mechanisms, detailing the cause of discarding

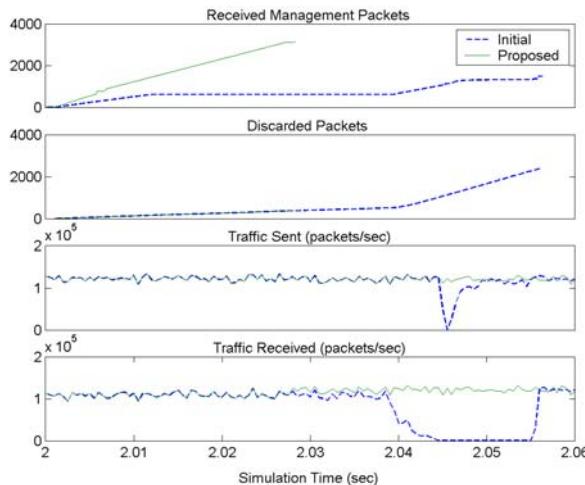
to reach their destination. This is the only cause of packet discarding for the proposed mechanism. Finally, the ‘Inactive Tables’ label refers to the packets that can not be injected into the network because the endpoint routing tables are empty (one of the steps in the static reconfiguration process [21]).

To conclude this evaluation, Fig. 5 illustrates the instantaneous behavior of both the initial and proposed mechanisms. To obtain these results, we have scheduled the activation (Fig. 5a) and the deactivation (Fig. 5b) of a switch in a 6×6 mesh, which is made up of 36 switches and 20 endpoints. In both cases, the topological change occurs at 2.0 secs. Note that the plots show the execution of the entire assimilation process. These conclusions are very similar for the rest of topologies evaluated.

For all plots, the x-axis represents the simulation time. The top plots represent the amount of management packets exchanged among the management entities. For the initial mechanism, the two steps in these plots corresponding to the topology discovery and the static reconfiguration phases, respectively. However, for the proposed mechanism, it is more difficult to appreciate the different phases.



(a) Activation



(b) Deactivation

Fig. 5. Instantaneous behavior of the evaluated management mechanisms for a 6×6 mesh

The second plots show the aggregate amount of data packets discarded during the change assimilation process. In case of a switch activation (Fig. 5a), the initial mechanism discards 1890 data packets and assimilates completely the change at 2.097 secs, whereas the proposed mechanism does not discard any data packet.

In case of a switch deactivation (Fig. 5b), for both initial and proposed mechanisms, the packet discarding starts immediately after the change detection occurs. As we can see in this figure, the initial discards data packets (2375

in total) until the reconfiguration phase finishes at 2.056 secs, and the proposed mechanism stop discarding data packets (372 in total) at 2.028 secs.

The bottom two plots in Fig. 5a and Fig. 5b show instantaneous network throughput, represented by the number of data packets sent/received per second in the whole fabric. For both cases, the initial management mechanism has a detrimental effect on the network service, and we can observe a gap in the instantaneous throughput plots, corresponding to the reconfiguration phase. Note that this gap is completely eliminated by using our proposed mechanism, and the network connectivity is recovered 0.3 secs before.

4 Conclusions and Future Work

This work collects in the same management mechanism efficient implementations for all the phases involved in the process of assimilating a topological change, for networks using source routing. First, the employment of the parallel discovery and distributed path computation algorithms speed up considerably the whole process. Moreover, the CGR reconfiguration technique allows a dynamic updating of the routing function. That is, the network is up and running during all the assimilation process. All the proposals can coexist and run simultaneously without interfering with each other.

Simulation results show that the combined mechanism significantly reduces the change assimilation time and, also, the amount of packets that are discarded during the process. From the point of view of upper-level applications, this scheme virtually eliminates the problem of reduced network service availability, which is characteristic of traditional mechanisms. As additional advantages, the proposed mechanism does not require the use of specific fabric resources to operate –it can easily be implemented in current commercial systems–, and its computation complexity is similar to previous proposals.

As future work, we plan to generalize the proposed implementation not to be dependent on up*/down* routing. Additionally, we want to adapt our proposals to distributed routing environments.

References

1. InfiniBand Trade Association: InfiniBand Architecture Specification Release 1.2 (2004), <http://www.infinibandta.org/>
2. Myrinet: Guide to Myrinet-2000 switches and switch networks, <http://www.myri.com/>
3. ASI-SIG: Advanced Switching Core Architecture Specification Revision 1.0 (December 2003), <http://www.picmg.org>
4. Rooholamini, M.: Advanced Switching: A new take on PCI Express (October 2004), <http://www.edn.com/article/CA468416.html>
5. Rooholamini, M., Kaapor, R.: Fabric discovery in ASI (October 2005), <http://www.networksystemsdesignline.com/>
6. Robles-Gómez, A., Bermúdez, A., Casado, R., Quiles, F.J.: Implementing the Advanced Switching fabric discovery process. In: Proceedings of the 7th Workshop on Communication Architecture for Clusters (CAC 2007) (March 2007)

7. Schroeder, M.D., Birrell, A.D., Burrows, M., Murray, H., Needham, R.M., Rodeheffer, T.L., Satterthwaite, E.H., Thacker, C.P.: Autonet: A high-speed, self-configuring local area network using point-to-point links. *IEEE Journal on Selected Areas in Communications* 9(8) (October 1991)
8. Casado, R., Bermúdez, A., Duato, J., Quiles, F.J., Sánchez, J.L.: A protocol for deadlock-free dynamic reconfiguration in high-speed local area networks. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 12(2) (February 2001)
9. Bermúdez, A., Casado, R., Quiles, F.J., Duato, J.: Fast routing computation on InfiniBand networks. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 17(3) (March 2006)
10. Robles-Gómez, A., Bermúdez, A., Casado, R., Solheim, Å.G., Sødring, T., Skeie, T.: A new distributed management mechanism for ASI based networks. *Computer Communications (COMCOM)* 32(2) (February 2009)
11. Lysne, O., Skeie, T., Reinemo, S., Theiss, I.: Layered routing in irregular networks. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 17(1) (January 2006)
12. Solheim, Å.G., Lysne, O., Skeie, T., Sødring, T., Theis, I.T., Johnson, I.: Routing for the ASI fabric manager. *IEEE Communications Magazine* 44(7) (July 2006)
13. Boden, N.J., Cohen, D., Felderman, R.E., Kulawik, A.E., Seitz, C.L., Seizovic, J.N., Su, W.: Myrinet: A gigabit-per-second local area network. *IEEE Micro.* 15(1) (February 1995)
14. Lysne, O., Duato, J.: Fast dynamic reconfiguration in irregular networks. In: *Proceedings of the 2000 IEEE International Conference of Parallel Processing (ICPP 2000)*, Toronto, Canada (2000)
15. Acosta, J.R., Avresky, D.: Intelligent dynamic network reconfiguration. In: *Proceedings of the 2007 International Parallel and Distributed Processing Symposium, IPDPS 2007* (March 2007)
16. Theiss, I., Lysne, O.: LORE - LOcal REconfiguration for fault management in irregular interconnects. In: *Proceedings of the 2004 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2004* (2004)
17. Pinkston, T., Pang, R., Duato, J.: Deadlock-free dynamic reconfiguration schemes for increased network dependability. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 14(8) (August 2003)
18. Lysne, O., Montañana, J.M., Fliech, J., Duato, J., Pinkston, T.M., Skeie, T.: An efficient and deadlock-free network reconfiguration protocol. *IEEE Transactions on Computers* 57(6) (2008)
19. Montañana, J.M., Fliech, J., Duato, J.: Epoch-Based Reconfiguration: Fast, simple, and effective dynamic network reconfiguration. In: *Proceedings of the 2008 International Parallel and Distributed Processing Symposium, IPDPS 2008* (April 2008)
20. Robles-Gómez, A., Bermúdez, A., Casado, R., Solheim, Å.G.: Deadlock-free dynamic network reconfiguration based on close up*/Down* graphs. In: Luque, E., Margalef, T., Benítez, D. (eds.) *Euro-Par 2008. LNCS*, vol. 5168, pp. 940–949. Springer, Heidelberg (2008)
21. Robles-Gómez, A., Bermúdez, A., Casado, R., Quiles, F.J., Skeie, T., Duato, J.: A proposal for managing ASI fabrics. *Journal of Systems Architecture (JSA)* 54(7) (July 2008)
22. Robles-Gómez, A., García, E.M., Bermúdez, A., Casado, R., Quiles, F.J.: A model for the development of AS fabric management protocols. In: Nagel, W.E., Walter, W.V., Lehner, W. (eds.) *Euro-Par 2006. LNCS*, vol. 4128, pp. 853–863. Springer, Heidelberg (2006)
23. OPNET: <http://www.opnet.com>

Dependability Analysis of a Fault-Tolerant Network

Reconfiguring Strategy

Vicente Chirivella¹, Rosa Alcover¹, José Flich², and José Duato²

¹ Department of Statistics and Operation Research
`{vchirive, ralcover}@eio.upv.es`

² Department of Information Systems and Computer Architecture
Universidad Politécnica de Valencia,
Camino de Vera 14, 46022 Valencia, Spain
`{jflich, jduato}@gap.upv.es`

Abstract. Fault tolerance mechanisms become indispensable as the number of processors increases in large systems. Measuring the effectiveness of such mechanisms before its implementation becomes mandatory. Research toward understanding the effects of different network parameters on the dependability parameters, like mean time to network failure or availability, becomes necessary. In this paper we analyse in detail such effects with a methodology proposed previously by us. This methodology is based on Markov chains and Analysis of Variance techniques. As a case study we analyse the effects of network size, mean time to node failure, mean time to node repair, mean time to network repair and coverage of the failure when using a 2D mesh network with a fault-tolerant mechanism (similar to the one used in the BlueGene/L system), that is able to remove rows and/or columns in the presence of failures.

1 Introduction

Compute-intensive applications require a huge amount of processing power. The required levels of computing can only be achieved with massively parallel systems, including thousands of processing elements. The huge number of processors and associated devices (memories, switches, and links, etc.) significantly affects the probability of failure. Each individual component can fail and, thus, the probability of failure of the entire system increases dramatically. In addition, failures in the interconnection network may isolate a large fraction of the machine, containing many healthy processors that otherwise could have been used. Although network components, like switches and links, are robust, they are working close to their technological limits and they are prone to failures. Increasing clock frequencies leads to a higher power dissipation, which again could lead to premature failures. Therefore, fault-tolerant mechanisms for interconnection networks are becoming a critical design issue for large massively parallel computers [1], [2], [3], [4], [5], [6].

In such large systems, knowing the effectiveness of a fault-tolerant mechanism becomes challenging. As the system increases in size, several isolated faults may coexist and their interaction can break down the entire system. The most direct method to evaluate system dependability is based on actual measurement of the system under

evaluation. However, the system is not available during the design phase, and hence, abstract models are necessary for reliability prediction.

Component redundancy is the easiest way to tolerate faults as components in the system are replicated and, once a failed component is detected, it is simply replaced by its spare. An enhanced version of this technique does not require spare components. It simply bypasses faulty components, together with some healthy components, to maintain network regularity. For instance, in the BlueGene/L project [7], the nodes are connected by using a 3D torus. This supercomputer is constituted by 65536 nodes, which are allocated over 64 racks of 1024-nodes, with two 512-node midplanes per rack [8]. In each rack, there are additional link boards that connect links from the backplane with connectors in the front panel so that each rack can be connected with neighbouring ones. Internal switches on the link boards allow a plane to be connected to the next one or to skip one plane (512 nodes). Once a failure is detected, all the nodes included in the midplane that contain the faulty node/link are marked as faulty and bypassed.

We have proposed a methodology [9] based on Markov chains to analyze the system dependability parameters. We used it to analyze the effect of a fault-tolerant routing algorithm on dependability parameters. In this paper, we apply this methodology on a fault-tolerant mechanism similar to the one used in the BlueGene/L supercomputer [7]. The study of the interconnection network reliability has been treated extensively in the literature. But we have not found any previous work to study the effect of the interconnection network design parameters on dependability parameters. This type of analysis quantifies the impact of these parameters. That is particularly relevant when the goal is to design an interconnection network with a given level of dependability. For this, the interconnection network designer must know which of these parameters are relevant, how they interact with each other, and the behaviour of the dependability parameters for different values of the design parameters.

The goal of our paper is to analyze the effect of different network design parameters on dependability parameters. In particular, we will analyze the mean time to network failure and the steady state availability. For the network parameters, we use the network size, the mean time to node failure, the mean time to node repair, the mean time to network repair and the coverage of the failure. We will focus on a system with nodes connected through a 2D mesh network. Each node will include a processor and a router.

The remaining of this paper is organized as follows. In Section 2 we summarize the necessary steps to apply our methodology. Then, in Section 3 we present the case study. Finally, in Section 4 we show the conclusions and future work.

2 Methodology

On a previous paper we proposed a methodology [9] to analyze the system dependability parameters. This methodology can be summarized in the following steps:

- 1- Define the Interconnection Network Fault Model. In this step the assumed hypothesis about network fault tolerance mechanism must be established.
- 2 - Select the network dependability parameters. This step requires the selection of the dependability parameters that will quantify the reliability characteristic that we want

to study. Some network dependability parameters are the reliability function, the mean time to network failure, the mission time, the availability function, the computation reliability, and more [10].

3 - Define the network states and the transition rates. Now, we must specify the Markov chains used for modelling the network dependability behaviour. For this, it is necessary to define the states that represent the network operation and to obtain the transition rates between them.

4 - Compute the values of the network dependability parameters. These values are computed by solving the system of differential equations obtained from transition rates [11]. This system can be easily solved by using the Laplace transform. The system solution provides the probabilities of being in each network state defined in the step 3. These probabilities allow us to compute the dependability parameters.

5 - Analyze the results. Finally, the obtained dependability parameters values must be analyzed.

3 A Case Study

In this paper we have studied the effect of different interconnection network design parameters on the network dependability when a network reconfiguring strategy is used as a fault tolerant method. This reconfiguration strategy consists on the elimination of the entire row or column that contains a faulty node. After the elimination, a small fault free sub-network is obtained and the task can be executed while there are enough working nodes. We have applied the previous methodology to a practical case, two dimensional meshes. In this section we explain the assumed hypothesis, the performed analysis and the obtained results in this type of network.

1- Define the Interconnection Network Fault Model.

The definition of the interconnection network fault model includes the network selection and the establishment of the assumed hypothesis about its operation. We have studied 2D mesh, with wormhole switching. Routing algorithm does not affect the dependability analysis since a recovery mechanism will provide a fault free mesh. We suppose that only nodes (processor and router) fail, not the links.

The fault tolerant method analyzed consists on mesh reconfiguration to obtain a fault free sub-mesh. The reconfiguration is carried out by means of the elimination of the row or column that contains the faulty node. We eliminate alternatively rows and columns so that the mesh has a quite square aspect, and the surviving nodes are relatively close to each others. We also consider that the elimination of two contiguous rows (or columns) affects the message propagation time on the channel, and therefore we assume that the system fails. This hypothesis is not applied if one of the rows (or columns) is on the mesh edge.

The system also fails if less than the 80% of its initial nodes are available. Once reached the limit, mesh performance is unacceptably low and should be repaired. This limit of 80% of healthy nodes has a direct effect on network reliability parameters, because the mesh is able to support more failures as its size increases. The method admits a new failure because the number of healthy nodes after eliminating the rows (or columns) is the same or larger than the elected limit, 80% of the original size.

A recovery mechanism recovers the system from the occurrence of node failures during operation. The recovery consists of the detection of the fault, the identification of the faulty node, the correction of the induced errors, and the elimination of the column (or row). However, fault detection mechanisms are not perfect, and they can fail with a certain probability. The probability of system recovery when a fault occurs is called coverage, C [12]. In this work, this probability is assumed to be constant. Finally, we also assume that the network is repairable. The failure of a node is repaired one at a time, and the network is fully repaired after its failure. When the interconnection network is repaired, it is completely repaired, replacing all the faulty components.

On the other hand, the modelling of the interconnection network has been based on continuous-parameter Markov chains. The network design parameters that we have selected for this study are the mesh size, the mean time to node failure, the mean time to node repair, the mean time to network repair and the coverage of the failure. We have considered exponential distribution with parameter λ for the node failure times, exponential distribution with parameter μ for the node reparation times, exponential distribution with parameter v for the network reparation times, and a uniform distribution of message destinations. The typical values for these design parameters are the followings. The mean time to node failure, ($MTTNF$, $1/\lambda$) is measured in months, its typical value being 2, 4, 6, 8, 10 and 12 months (named F1 to F6). The lowest value corresponds to machines mounted in rooms and wired externally, while the highest one corresponds to multicomputers assembled in cabinets. The mean time to node repair ($MTTNR$, $1/\mu$) is measured in hours. We have considered 3, 6, 12 and 24 hours (named R1 to R4). The lowest value corresponds to critical applications, while the highest value corresponds to non-critical applications. The mean time to network repair ($MTTNWR$, $1/v$) is also measured in hours, although their values are in the order of days to repair. We have selected 72, 720 and 2160 hours (named D1 to D3). Two coverage values have been used, 0,99 and 0,999 (C1 and C2 respectively). Finally, the meshes analyzed are square, from 4x4 to 24x24 nodes. This is a small number of nodes in comparison with real machines, and it is due to the limitations of power computation available nowadays.

2 - Select the network dependability parameters.

We must select the dependability parameters [10] that will quantify the reliability characteristic that we want to study. They are the $MTTNWF$ and the steady state availability. The $MTTNWF$ is the mean time at which the network first fails. If we integrate the reliability function we compute the $MTTNWF$. This function provides the probability that the network works correctly until time t given that it was operational at time 0. The steady state availability is the long term availability, the limit value of the instantaneous availability function when time approaches infinity. The instantaneous availability function is the probability that the network is operating correctly and it is available to work at the instant of time t .

3 - Define the network states and the transition rates.

We must specify the Markov chains used for modelling the network dependability behaviour, that is, we must define the network states and the transitions between them. According to dependability parameters and the number of faulty network nodes,

we have defined the following states in the models: Correct State, the original mesh with all their nodes working; Degraded State, when a node fails, but the fault detection mechanisms eliminate a row (or a column) and the mesh continues working, and Failed State, when the failure of the network occurs, whatever its cause is. After a sojourn in a state, the state of the Markov chain will make a transition to another state. The destination state depends on the network functioning and the dependability parameter computed. With the previous states, we propose the models to compute the interconnection network reliability and availability. These models depend on the number of faulty nodes supported, that depends in turn on the mesh size, as we have seen previously. Therefore the models will differ in the number of degraded states (number of faults).

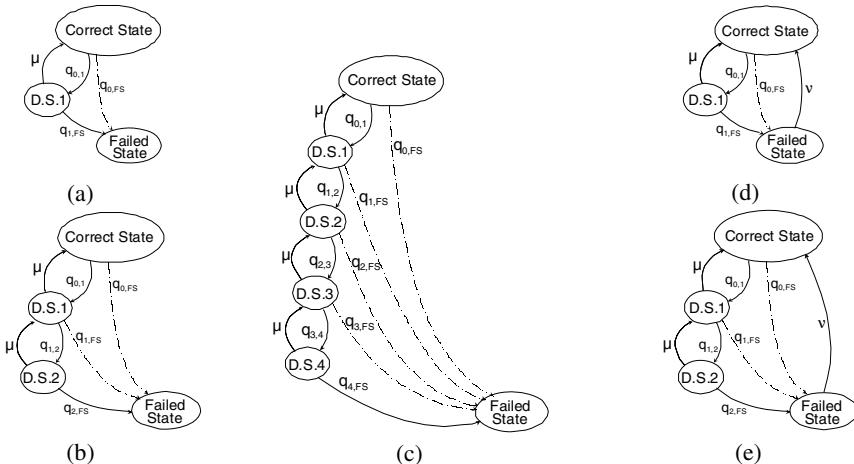


Fig. 1. Examples of State Diagrams to model network dependability behavior. Models to compute MTTNWF (a) and AVAILABILITY (d) for a 5x5 to 9x9. A second pair of figures for 10x10 to 14x14 meshes, MTTNWF (b) and AVAILABILITY (e). Model (c) is used to compute MTTNWF for 19x19 to 23x23 meshes.

The model on Fig. 1(a) is used to study the MTTNWF on meshes from 5x5 to 9x9 nodes. These networks support a single failure, represented as the degraded state D.S.1. If we add a transition between the Failed State and the Correct State, that is, the network is repaired (MTTNWR), Fig. 1(d) then the model will allow us to compute network availability. A large mesh will support more faults and consequently new degraded states must be added to the model. Meshes whose size are from 10x10 to 14x14 nodes support two faults, and a second degraded state, D.S.2, appears in Fig. 1(b) to compute MTTNWF and Fig. 1(e) for availability. Finally, Fig. 1(c) models the MTTNWF for meshes from 19x19 to 23x23 nodes, which support four failures.

The network starts in the Correct State and evolves to the Degraded State when a fault occurs, the failure is covered and there are not adjacent faults. The network changes to the Failed State if at least one of the two conditions is met: the recovery mechanism fails or there are two eliminated adjacent rows or columns. From a Degraded State, the network can reach another Degraded State until network failure, a

previous Degraded State if the node is repaired, or Failed State if the fault is not covered. On the state diagram to compute availability there is another transition, from Failed State to the Correct State when the network is repaired.

In this study we assume that the system fails when the number of healthy nodes reaches 80% of its initial number, or when a new failure produces two adjacent columns (or rows). Therefore we must analyze the effect of faulty nodes (pairs, trios, ...) positions on the mesh. Some trios of faulty nodes produce the mesh failure if they are in two adjacent columns, and others do not. We must quantify the proportion of faulty node combinations that produces the mesh failure. Thus, the transition rates between states depend on the size of the mesh and the position of the faulty nodes.

Once the failure combinations have been obtained, the transition rates between states can be computed, given the mesh size. The expression of the transition rate $q_{i,i+1}$ from the Degraded State i ($i \geq 0$) to the next Degraded State $i+1$ and the transition rate $q_{i,FS}$ to the Failed State, F.S., are given by

$$q_{i,i+1}(t) = (N - i) \cdot rp_{i+1} \cdot C \cdot \lambda \quad (1)$$

$$q_{i,FS}(t) = (N - i) \cdot (1 - rp_{i+1} \cdot C) \cdot \lambda \quad \text{with } rp_{i+1} = nfc_{i+1} / (nfc_i \cdot (N - i))$$

where C is the coverage; λ is the node failure rate; N is the number of nodes in the mesh; rp_{i+1} is the probability of network working when a new failure takes place, knowing that i failures have already taken place in an operational network; and nfc_{i+1} is the number of combinations of $i+1$ faulty nodes that do not cause the network failure, knowing that such combinations are obtained from the combinations of i faulty components that did not cause the network failure, with the position of a new faulty node. The number of combinations to study grows exponentially with the mesh size, and makes that this calculation method can only be applied to small meshes.

We present, as an example, the expressions of the transition rates for a 20x20 mesh. On the 20x20 mesh there are 400 nodes. The failure of any node produces the elimination of a column. There are no adjacent eliminated columns so the resulting mesh works. Transition rates are $q_{0,1} = 400\lambda C$ and $q_{0,FS} = (400 - 400\lambda C)\lambda$. If a new failure takes place, at any of the 380 remainder nodes (20x19), a row containing the faulty node is eliminated. Again there are no adjacent eliminated rows at the $400 \cdot 380 = 152000$ pairs of faulty nodes positions. Transition rates are $q_{1,2} = 380\lambda C$, $q_{1,FS} = (380 - 380\lambda C)\lambda$ and $q_{1,0} = \mu$. The mesh reduces its size to 19x19, 361 nodes. Now with a new node failure it is possible that some of the $400 \cdot 380 \cdot 361 = 54720000$ trios of faulty nodes produce two adjacent eliminated columns, and therefore the failure of the mesh. In fact 49384800 trios do not produce the mesh failure, so the transition rate from Degraded State 2 to Degraded State 3, $q_{2,3}$, is $49384800 / 152000 = 324,9$. Transition rates are $q_{2,3} = 324,9\lambda C$, $q_{2,FS} = (361 - 324,9\lambda C)\lambda$ and $q_{2,1} = \mu$. The next transition rate, $q_{3,4}$ is $15200641440 / 49384800 = 307,8$, so $q_{3,4} = 307,8\lambda C$, $q_{3,FS} = (334 - 307,8\lambda C)\lambda$ and $q_{3,2} = \mu$. Finally with the fifth failure the system fails because it reaches the 80% limit, and transition rates are $q_{4,FS} = 324\lambda C$ and $q_{4,3} = \mu$. We should note that the verification of each of the 54720000 trios requires a large computation time.

4 - Compute the values of the network dependability parameters.

In this step the values of the MTTNWF and the steady state availability are obtained. For this, we must solve the system of differential equations that govern the state

probabilities [11]. We have used the Laplace transform to solve the system and to obtain the unknown state probabilities.

Both the reliability and the availability functions are the sum of the probabilities of being in one operational state at each instant of time. They are the sum of the probabilities of being in the Correct State or in a Degraded State, computed on their respective state diagrams. We have used the integration properties of the Laplace transform to compute the MTTNWF, and the final value theorem of the Laplace transform to compute the steady state availability.

5 - Analyze the results.

Next, we use the ANOVA [13] to analyse the two dependability parameters selected, the MTTNWF and the steady state availability. The ANOVA is a powerful statistical tool for studying the effect of factors on the average value of the response variable. Once calculated the ANOVA table, we determine the effects that affect each dependability parameter. This must be done using the Mean Square column instead of the F-ratio or p-Value, as the dependability parameters have been obtained through an analytical model and they are not random variables. An effect could be considered a major effect if its Mean Square is large, or it is larger than the Mean Square of other effects. Another way for measuring the importance of a factor, or interaction, is to calculate the percentage of the dependability parameter variability that explains. This is made dividing the sum of squares of this factor (or interaction) over the total sum of squares. Only the major effects or interactions are worth to analyze in depth, and it could be done using the Mean Plots. Due to the lack of available space in this paper the ANOVA tables have been omitted, and only the most outstanding results are commented.

5.1 - Mean time to network failure.

Now we analyze the effect of the design parameters on the MTTNWF. The proposed model takes into account the design parameters and its double and triple interactions. It has been necessary to use the logarithmic transformation on the response variable (MTTNWF) to obtain an error with normal distribution and to guarantee the compliance of the ANOVA hypothesis.

Once computed the ANOVA, the Mean Squares for all the simple effects as well as some double have a large value. This means that they clearly affect the MTTNWF. Nevertheless some parameters are much more important than others. A factor is more important than other if its Mean Square is among highest. We can observe that the main parameter is the MTTNF, followed by the MTTNR. With a quite smaller importance we have the effect of the mesh size and the interaction between the MTTNR and the coverage. All they together are able to explain the 95,1% of the observed variability in the MTTNWF. We can highlight the fact that coverage is not an important factor by itself, although it is part of the most important interaction. As an advance of the results that we will present on Fig. 3, Fig. 2(a) shows the increment in MTTNWF for the most outstanding effects. As can be easily deducted, MTTNF has a larger impact on MTTNWF than MTTNR (the quality of the node is more important than the repair time of the node). On the other hand, MTTNWF is largely increased when the MTTNF is increased (the quality of the node is increased). However, this increase is outstanding only if the repair time is short (R1, 3 hours).

If repair time is large (R4, 24 hours), then MTTNF is not so outstanding (the quality of the node can be lower). Finally, we can see that the coverage helps increasing the MTTNWF only if the MTTNF is high (FN6, 12 months) or the repair time is short (R1, 3 hours).

Let's now look in detail the previous conclusions, the most important effects on the MTTNWF. This dependability parameter increases when increasing the MTTNF. This is quite obvious, because as much as you improve the nodes life, the more time the mesh will work. This design parameter explains 46,7% of the observed variability in the MTTNWF, and improves its value in 1897 hours in the MTTNF range selected.

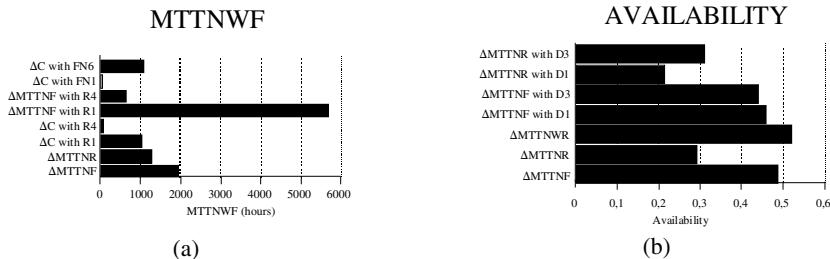


Fig. 2. MTTNWF (a) and Availability (b) versus main effect or interaction

The MTTNWF diminishes when increasing the size of the mesh (Fig. 3(a)). This is also very simple of explaining, since the more nodes in mesh the larger is the probability that at least one of them (and the mesh) fails. Mesh size explains the 28,2% of the observed variability of the MTTNWF, and the difference between the two end limits is 4039 hours. In spite of this great value, we must remember that the mesh size is determined by the needs of computing power and not for a desired value of the MTTNWF. According to the nature of the fault tolerant method analyzed, the MTTNWF increases punctually at certain mesh sizes. The increase takes place when the method admits a new failure in the mesh because the number of nodes that remain healthy is the same or larger than the 80% node limit. In the range of selected mesh sizes the increase takes place at 10x10, 15x15, 19x19 and 24x24 nodes. When the mesh size reaches this limit a new failure is admitted, and produces the improvement of the MTTNWF. After this punctual improvement, the MTTNWF diminishes quickly with the mesh size because there are more and more nodes to fail. It can also be observed that the increases of the MTTNWF at the limits are smaller with the size of the mesh. Although a new failure can be admitted, the number of nodes (exposed to failures) in the mesh is so high that it counteracts the improvement in the MTTNWF.

The decrease of the MTTNR has a larger effect as the coverage increases Fig. 3(b). As the coverage increases, the probability of being in an operative state also increases, and therefore there are more opportunities to network repair. The effect of increasing the coverage for a long time to node repair is 9 hours, but if the node is repaired quickly, the effect could be 992 hours.

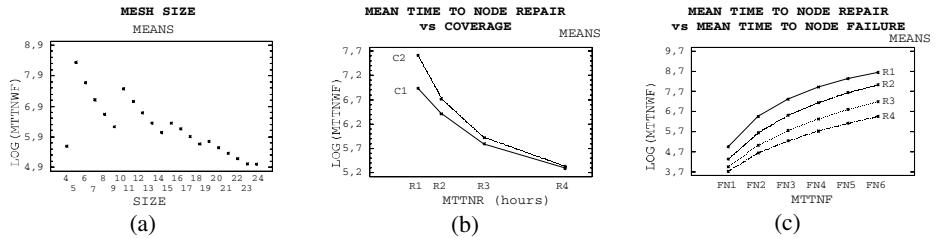


Fig. 3. Mean Plots of MTTNWF logarithm versus mesh size (a) and main double interactions: MTTNF-coverage (b) and MTTNF-MTTNR (c)

The MTTNWF increases with the MTTNF and decreases with the MTTNR (Fig. 3(c)). But the increase of the MTTNWF with the MTTNF is largest as the lower is the MTTNR. An increase of the MTTNF has a larger effect if the node is quickly repaired, or a decrease of the MTTNR has a larger effect when the node fails slower. The effect of increase the MTTNF when the node is repaired slowly is 593 hours, but if the node is repaired quickly its effect could be up to 5625 hours.

Although the studied mesh size is very small in comparison with the implemented multicomputers, it is possible to extrapolate the conclusions obtained for larger meshes. The MTTNWF for meshes whose size are larger than 24x24 will be lower than 149 hours. Its punctual increase at the 80% size limits will be negligible with the increase of the mesh size. All the interactions of the mesh size with the main effects are significant, but no one of them are important in comparison with the main effects. This means that the MTTNF, the MTTNR and the coverage, as well as the interaction between the MTTNR and the coverage, are the most important effects to explain the MTTNWF for larger meshes too.

5.2 - Steady state availability.

Next we study the effects of the design parameters on the steady state availability, or availability to simplify. Availability is the proportion of time that the network is working or waiting to work.

As an advance of the results of the availability analysis, Fig. 2(b) shows the effect on availability for different factors. The factors that help increasing mostly network availability are MTTNWR and MTTNF, that is, a short network repair time and a large node failure time. However, having a small node repair time does not help much in the availability of the network. For interaction between factors, we can see that benefits from MTTNR and MTTNF are small when using short (D1, 72 hours) and large values (D3, 2160 hours) of MTTNWR.

According to the ANOVA performed, the Mean Square column shows us that the main parameter is the MTTNWR. With a quite smaller importance we have the effect of the MTTNF and MTTNR, followed by mesh size and finally the coverage. The remainder interactions have a scarce effect on availability. In fact the first four design parameters explain the 90,8% of the variability observed in the network availability, and the MTTNWR explains by itself the 41,2% of the availability.

Network availability increases with the MTTNF. When increasing the MTTNF the MTTNWF increases too, and then the proportion of time that the network can be used

increases. The MTTNF explains the 24,4% of the variability observed in the availability, and increases mesh availability in 47,9%.

The availability increases with the decrease of both the mean time to node and network repair. The less time is used in repairing a single node the long the network will be working or ready to work, and its availability increases. This parameter explains the 10,5% of the variability, with an increase of 28,6%. The less time is used to repair the whole network the sooner will be ready to work and its availability also increases. This is the most important parameter, with a 41,2% of explanation and a 51,2% of increase of availability.

Now we present on Fig. 4 the Means Plots for the mesh size effects, and the main two interactions. With these plots we will be able to observe the importance and the evolution of network availability with the design factors.

Network availability diminishes as the mesh size increases. The availability diminishes quickly with the mesh size because of the large number of nodes to fail. But, as we can see on Fig. 4(a), there are punctual increases of availability. This is due to the nature of the fault tolerant method analyzed (as it happened to the MTTNWF). When the mesh sizes reaches this limit a new failure is admitted, it produces the increase of the working time and therefore the availability. This punctual increase of the availability diminishes with the size of the mesh. Although a new failure can be admitted, the number of nodes in the mesh is so high that it hardly produces an improvement of availability. The mesh size explains the 14,7% of the observed variability on the network availability, and its effect can be up to 47,1% of availability.

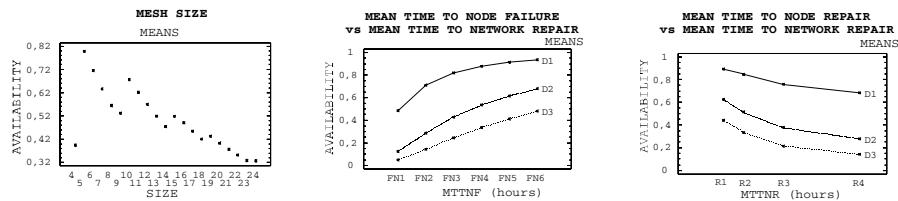


Fig. 4. Means of availability versus mesh size (a), and main double interactions: MTTNF-MTTNWR (b) and MTTNR - MTTNWR (c)

The increase of the availability with the MTTNF is larger as smaller is the MTTNWR (Fig. 4(b)), but becomes stable for small MTTNWR. This means that the advantage to improve the MTTNF loses importance if the mesh is repaired quickly.

The increase of the availability with the decrease of the MTTNR is larger as larger is the MTTNWR (Fig. 4(c)). For long MTTNWR the decrease of the MTTNR produces a larger availability increases than for short MTTNWR. Now the advantage to improve the MTTNR loses importance if the mesh is repaired quickly.

If we try to extrapolate the results obtained for larger meshes, the availability for meshes whose size are larger than 24x24 will be lower than 32,6%. Its punctual increase at the size limits will be negligible with the increase of the mesh size. Although most of the interactions of the mesh size with the main effects are significant, no one of them are important in comparison with the main effects. Therefore the MTTNF, the MTTNR, the MTTNWR and the coverage are the most important effects to explain the network availability for larger meshes.

4 Conclusions

The MTTNWF depends mainly on the MTTNF. We must select carefully the quality of the node components if we want to reach a given network lifetime. The second parameter is the MTTNR. A quick node repair will produce a considerable increase of the network lifetime. Other remarkable factors are the coverage and the interaction between the MTTNR and the coverage. The MTTNWF increases with the coverage, and the effect of reduce the MTTNR is larger if the coverage is high. Therefore we get an additional effect when increase simultaneously the coverage and the speed in the node repair. Something similar, but of smaller importance, happens with the MTTNF. The simultaneous increase of the coverage and MTTNF produces an additional increase in the network lifetime. Finally, there is an additional improvement of the network lifetime when we increase the node lifetime and when we diminish its time to repair. Fortunately, the improvement of all design parameters produces the improvement of the MTTNWF.

Another important design parameter is the mesh size. The drop of the MTTNWF with the mesh size is huge, but this parameter is selected to obtain some computing power level and not some reliability level. MTTNWF punctual increase at the size limits (80% of healthy nodes) will be negligible with the increase of the mesh size. The MTTNF, the MTTNR and the coverage, as well as the interaction between the MTTNR and the coverage, are the most important effects to explain the MTTNWF for larger meshes.

The network steady state availability depends mainly on the MTTNWR. A quick network repair increases considerably its availability, and therefore, if we need a high network availability we must repair it as soon as possible. A light improvement of this parameter is considerably more important than any improvement the other design parameters. The next two parameters are the MTTNF and the MTTNR. We have observed that the improvement of network availability when having more reliable nodes is higher if the network is quickly repaired, and that the decrease of the time to node repair has a larger effect if the network is quickly repaired too. However, we should point out that if the network is repaired really fast the effects of these factors are not longer important. Finally the improvement of each design parameters has a combined and positive effect on network availability.

Network availability drops dramatically with the mesh size. As in the MTTNWF the increase at the size limits will be negligible with the increase of the mesh size. The MTTNF, the MTTNR, the MTTNWR and the coverage continue being the most important effects to explain the network availability for larger meshes.

Finally we briefly compare these results with our previous work [9]. We compute the same dependability parameters using a fault-tolerant routing algorithm, as the way of enhancing mesh reliability. The results are clearly better for the routing algorithm. This conclusion should be analyzed in detail, but generally speaking it is correct. Therefore, the way we manage the elimination of the failed component is a decisive factor on network reliability.

As a future work we plan to apply our methodology on new fault-tolerant mechanisms used in networks on chip. As technology advances more components are placed within a chip and a small network inside the chip is required. With the help of this methodology we will quantify the effects of design parameters on the lifetime of the entire chip taking into account the effectiveness of the fault-tolerant mechanism used.

Acknowledgement

This work has been jointly supported by the Spanish MEC and the European Commission FEDER funds under grants Consolider-Ingenio 2010 CSD2006-00046 and TIN2006-15516-C04-01.

References

- [1] Ho, C.T., Stockmeyer, L.: A New Approach to Fault-Tolerant Wormhole Routing for Mesh-Connected Parallel Computers. *IEEE Trans. Computers* 53(4), 427–439 (2004)
- [2] Wu, J.: A Fault-Tolerant and Deadlock-Free Routing Protocol in 2D Meshes Based on Odd-Even Turn Model. *IEEE Trans. Computers* 52(9), 1154–1169 (2003)
- [3] Jiang, Z., Wu, J., Wang, D.: A New Fault Information Model for Fault-Tolerant Adaptive and Minimal Routing in 3-D Meshes. In: Proc. Int'l Conf. Parallel Processing, June 2005, pp. 500–507 (2005)
- [4] Zhou, J.P., Lau, F.C.M.: Multi-Phase Minimal Fault-Tolerant Wormhole Routing in Meshes. *Parallel Processing* 30(3), 423–442 (2004)
- [5] Puente, V., Gregorio, J.A., Beivide, R., Vallejo, F.: A Low Cost Fault-Tolerant Packet Routing for Parallel Computers. In: Proc. Int'l Parallel and Distributed Processing Symp. (April 2003)
- [6] Puente, V., Gregorio, J.A., Vallejo, F., Beivide, R.: Immunet: A Cheap and Robust Fault-Tolerant Packet Routing Mechanism. In: Proc. Int'l Symp. Computer Architecture, June 2004, pp. 198–211 (2004)
- [7] The BlueGene/L Team, An Overview of the BlueGene/L Supercomputer. In: Proc. ACM/IEEE Conf. Supercomputing, November 2002, pp. 1–22 (2002)
- [8] Gara, A., et al.: Overview of the Blue Gene/L System Architecture. *IBM J. Research & Development* 49(2/3), 195–212 (2005)
- [9] Chirivella, V., Alcover, R.: A New Reliability Model for Interconnection Networks. In: Bode, A., Ludwig, T., Karl, W.C., Wismüller, R. (eds.) Euro-Par 2000. LNCS, vol. 1900, pp. 909–917. Springer, Heidelberg (2000)
- [10] Beaudry, M.D.: Performance-Related Reliability Measures for Computing Systems. *IEEE Transactions on Computers* 27(6), 540–547 (1978)
- [11] Bolch, G., Greiner, S., de Meer, H., Trivedi, K.S.: Queueing Networks and Markov Chains. Wiley Interscience, Hoboken (1998)
- [12] Dugan, J.B., Trivedi, K.S.: Coverage Modeling for Dependability Analysis of Fault-Tolerant Systems. *IEEE Transactions on Computers* 38(6), 775–787 (1989)
- [13] Scheffé, H.: The Analysis of Variance. Willey, New York (1959)

RecTOR: A New and Efficient Method for Dynamic Network Reconfiguration

Åshild Grønstad Solheim, Olav Lysne, and Tor Skeie

Networks and Distributed Systems Group, Simula Research Laboratory
Lysaker, Norway
Department of Informatics
University of Oslo
Oslo, Norway

Abstract. Reconfiguration of an interconnection network is fundamental for the provisioning of a reliable service. Current reconfiguration methods either include deadlock-avoidance mechanisms that impose performance penalties during the reconfiguration, or are tied to the Up*/Down* routing algorithm which achieves relatively low performance. In addition, some of the methods require complex network switches, and some are limited to distributed routing systems. This paper presents a new dynamic reconfiguration method, RecTOR, which ensures deadlock-freedom during the reconfiguration without causing performance degradation such as increased latency or decreased throughput. Moreover, it is based on a simple concept, is easy to implement, is applicable for both source and distributed routing systems, and assumes Transition-Oriented Routing which achieves excellent performance. Our simulation results confirm that RecTOR supports a better network service to the applications than Overlapping Reconfiguration does.

1 Introduction

Reliable interconnection networks  are essential for the operation of current high-performance computing systems. An important challenge in the effort to support a reliable network service is the ability to efficiently restore a coherent routing function when a change has occurred in the interconnection network's topology. Such a change in topology could be a result of an unplanned fault in one of the network's components, and, as the size of systems grow, the probability of a failing component increases. Furthermore, planned system updates, where network components are removed or added, could also cause changes in the topology. Regardless of the cause of the topology change, the disturbance of the network service provided to the running applications should be minimized.

When a change has occurred, a new routing function must be calculated for the resulting topology, and we refer to the transition from the old routing function to the new one as *reconfiguration*. A main challenge related to reconfiguration is deadlock-avoidance. The transition from one routing function to another may result in deadlock even if each routing function is deadlock-free, as packets that

belong to one of the routing functions may take turns that are not allowed in the other [2].

Most *static reconfiguration* methods [3,4,5,6] do not allow application traffic into the network during the reconfiguration. This has an obvious negative impact on the network service availability, but eliminates the risk of deadlock, as packets routed according to only one of the routing functions are present in the network at a time.

A number of studies [7,8,9,10,2,11,12,13,14,15] have focused on *dynamic reconfiguration*, which allows application traffic into the network during reconfiguration, and thereby aims at supporting a better network service than static reconfiguration do. The studies listed above present general approaches that are not tied to particular network technologies. Other approaches target a specific technology such as InfiniBand [16] (see e.g. [17,18]).

Partial Progressive Reconfiguration (PPR) [8] and Close Graphs (CG) [15] are based on the Up*/Down* [3] routing algorithm, and both restore a correct Up*/Down* graph from a graph that has been rendered incorrect by a topology change. PPR changes the direction of a subset of the network links through a sequence of partial routing table updates, and is only useful in distributed routing systems. CG restricts the new Up*/Down* graph such that packets belonging to the old and new routing functions can coexist in the network without causing deadlocks, and thereby supports an unaffected network service during reconfiguration. Neither PPR nor CG requires virtual channels (VCs) to achieve deadlock-freedom. However, both methods depend on complicated procedures to establish the new routing function. Furthermore, the Up*/Down* routing algorithm achieves relatively low performance (Section 3 gives a brief description of Up*/Down* and its performance issues).

Double Scheme [10] avoids deadlock during reconfiguration by utilizing two sets of VCs in order to separate packets that are routed according to the two routing functions. Each set of VCs accepts application traffic in turn while the other is being drained and reconfigured. Double Scheme can be used between any pair of routing algorithms. However, in order to avoid deadlock, Double Scheme generally requires a number of available VCs that resembles the sum of the number of VCs required by the old and new routing functions.

Overlapping Reconfiguration (OR) is perhaps the most efficient of the proposed methods to reconfigure an interconnection network. It has been categorized both as a dynamic reconfiguration scheme [19] and as a static reconfiguration scheme with overlapping phases [20]. OR can be used between any pair of routing algorithms, ensures in-order packet delivery, and does not depend on the availability of VCs. Originally, OR could only be applied in distributed routing systems, but an adaptation was recently proposed for source routed systems [21]. OR requires relatively complex network switches as each switch must hold and process information regarding the reception and transmission of tokens (special packets used for deadlock-avoidance). The tokens regulate the forwarding of packets that are routed according to the new routing function, and this regulation causes increased latency and reduced throughput during the reconfiguration.

Furthermore, OR demands that two sets of routing tables are kept during the reconfiguration.

This paper presents RecTOR, a new dynamic reconfiguration method which does not impose performance penalties during the change-over from one routing function to another. RecTOR is based on a simple principle that ensures deadlock-freedom while packets that follow either routing function can coexist in the network without restrictions. It does not require complex network switches, does not need more VCs than a routing function does, and is useful for both source and distributed routing systems. RecTOR assumes Transition-Oriented Routing (TOR) [22], a topology agnostic¹ routing algorithm that, given sound path selection, matches the performance of the topology specific Dimension-Order Routing (DOR) in meshes and tori. Our performance evaluation shows that, when compared to OR, RecTOR supports a superior network service during the reconfiguration.

As OR is used in the performance evaluation, the algorithm is detailed in Section 2. RecTOR is based on TOR, which is outlined in Section 3 through a comparison with Up*/Down* (which is also used in the performance evaluation). RecTOR is presented in Section 4, and its performance is evaluated in Sections 5 and 6. Section 7 presents our conclusions.

2 The OR Algorithm

OR uses a special packet called a *token* to prevent that deadlock occurs during the transition from one deadlock-free routing function, R_{old} , to another, R_{new} . A packet must be routed from source to destination according to only one of the routing functions. Let us refer to packets that follow R_{old} and R_{new} as packets_{old} and packets_{new}, respectively. OR uses the token to separate packets_{old} and packets_{new} on each (physical or virtual) communication channel, such that each channel first transmits packets_{old}, then the token, and finally packets_{new}.

The token propagation procedure is as follows:

- Each *injection channel* inserts a token between the last packet_{old} and the first packet_{new}.
- A *switch input channel* routes packets according to R_{old} until the token is processed, and thereafter routes packets according to R_{new} . After having processed the token, an input channel must only forward packets to output channels that have transmitted the token.
- A *switch output channel*, c_o , must not transmit the token until all input channels, c_i , for which dependencies² exist according to R_{old} from c_i to c_o , have processed the token.

For further details, see [20].

¹ A topology agnostic routing algorithm does not presuppose a particular topology.

² If a packet may use a channel c_b immediately after a channel c_a there is a channel dependency from c_a to c_b .

3 TOR versus Up*/Down*

Both TOR and Up*/Down* assign up and down directions to all the links in the network to form a directed acyclic graph (DAG) rooted in one of the switches.³ The path selection is, however, different for the two algorithms.

According to [23] deadlocks cannot form if cyclic channel dependencies are prevented. Up*/Down* breaks all cycles by prohibiting the turn from a down-link to an up-link. VCs are not required. With Up*/Down* all other switches/endnodes⁴ can reach the root (the only switch with no outgoing up-links) following one or more up-links, and the root can reach all other switches/endnodes following one or more down-links.

TOR selects paths without regard to the underlying DAG. For TOR, the purpose of the DAG is solely to identify the *breakpoints* – the turns where the cycles must be broken. As for Up*/Down*, the breakpoints are the down-link to up-link turns. TOR prevents deadlock by requiring that when a packet crosses a breakpoint (traverses from a down-link to an up-link), it makes a transition to the next higher VC. Thus, TOR supports a flexible shortest path routing, and can achieve significantly higher performance than Up*/Down* achieves. A main drawback of Up*/Down* is that the area around the root tends to become a hotspot. In addition, a legal route from one switch/endnode to another may be significantly longer than the shortest path in the physical topology.

For further details on TOR and Up*/Down*, see [22] and [3], respectively.

4 RecTOR

RecTOR is based on the following observations concerning TOR. During the operation of a network, changes in the topology can be reflected in the DAG. TOR selects paths independently of the underlying DAG (which sole purpose is to define the breakpoints that decide VC-transitions). Thus, after a topology change, a set of new paths that restores connectivity can always be found, provided that the topology is still physically connected.

Assume that G_{old} and G_{new} are the DAGs that apply before and after a topology change, respectively. In order to ensure that packets routed according to an old and new routing function can coexist in the network without causing deadlock, RecTOR makes only one assumption on the evolution of the DAG: *No breakpoint must be moved* during the transition from G_{old} to G_{new} . Thus, an edge that persists from G_{old} to G_{new} must keep its (up or down) direction. However, breakpoints (and turns in general) can be removed or added as vertexes and edges are removed or added, respectively.

Assume that TOR is used, and that a deadlock-free routing function, R_{old} (which includes VC-transitions according to G_{old}), applies when an unplanned

³ Links and switches are represented in the DAG by edges and vertexes, respectively.

⁴ An endnode is a compute node that generates and processes packets.

or planned topology change occurs.⁵ Then, RecTOR prescribes the following procedure to reconfigure the network:

1. Update the underlying DAG to reflect the change in topology. If a link or switch was removed, simply remove the corresponding edge or vertex (including its connecting edges), respectively. If a link or switch was added, add an edge or vertex (including its connecting edges), respectively. Avoid the introduction of cycles when assigning directions to newly added edges (see the Up*/Down* method).
2. Let TOR calculate a new deadlock-free routing function, R_{new} as follows: First, select a new set of paths which restore connectivity. Then, for each path, insert a transition to the next higher VC wherever the path crosses a breakpoint in G_{new} .
3. R_{new} can be applied instantly – packets routed according to R_{old} and R_{new} (or both) can coexist in the network without causing deadlock.

With regard to step 3 above, there is a risk of a packet looping if the packet are routed according to R_{old} in some of the switches and R_{new} in others (which could happen if the system uses distributed routing). Such looping could cause packet loss, as a packet that has reached the highest available VC and still needs to make another VC-transition must be rejected. A simple approach that prevents this problem implies that each switch holds routing tables for both R_{old} and R_{new} during the reconfiguration, and that each packet is tagged to indicate which of the routing functions should be used. However, a better solution could be adopted from the Internet research community, where several studies (e.g. [24]) have focused on preventing packets from looping during the update of routing tables. Like e.g. Double Scheme, RecTOR cannot guarantee in-order packet delivery during reconfiguration.

As deadlock avoidance is an inherent challenge in dynamic reconfiguration, we include and prove Lemma 1.

Lemma 1. *RecTOR provides deadlock-free reconfiguration.*

Proof. The proof is by contradiction. Assume that a deadlocked set of packets, S_d , is a set of packets where none of the packets can advance before another packet in the set advances. Assume also that a reconfiguration from R_{old} to R_{new} , where RecTOR is applied, results in some non-empty S_d .

Both R_{old} and R_{new} are, by themselves, deadlock-free. Thus, S_d must include at least one packet that is taking a turn which is present in both G_{old} and G_{new} , and which is either a breakpoint in G_{old} and not a breakpoint in G_{new} , or a breakpoint in G_{new} and not a breakpoint in G_{old} . In either case some breakpoint must have been moved during the transition from G_{old} to G_{new} , which contradicts the premise of RecTOR. \square

⁵ The change detection mechanism is outside the scope of RecTOR.

5 Experiment Setup

In order to compare the performance of RecTOR with the performance of OR, we conducted a number of experiments where a link fault and, subsequently, a switch fault were introduced and handled by reconfiguration. OR can be used between any pair of routing algorithms, and we consider two different alternatives. In the first case (referred to as OR_{TOR}), TOR is used. In the second case (referred to as $OR_{DOR/UD}$), DOR is used initially (for the fault-free topology) whereas $Up^*/Down^*$ is used after the first network component has failed.

Whereas TOR and $Up^*/Down^*$ are topology agnostic routing algorithms, DOR only works for fault-free meshes and tori. DOR avoids deadlock by first routing a packet in the X-dimension until the offset in this dimension is zero. Thereafter the packet is routed in the Y-dimension until it reaches its destination. For a torus topology, DOR needs two VCs for deadlock avoidance [23].

TOR can calculate shortest path routes in a number of different ways. In these experiments an out-port in the X-dimension is preferred over an out-port in the Y-dimension in every intermediate switch. For a fault-free mesh or torus, this gives similar paths as DOR, although the VC-use is different for many of the paths. In these experiments, the switch in the upper left corner of the mesh is the root of the $Up^*/Down^*$ graph.

The simulator model was developed in the J-Sim [25] environment. We consider both mesh and torus topologies⁶ of size 8×8 and 16×16 . In our experiments one endnode is connected to every switch. The packet size is 256 bytes, and both an ingress and egress buffer of a switch port can hold 6 packets per VC. The model applies virtual cut-through switching and credit-based flow control. A transmission queue in an endnode has space for 12 packets per VC, and overflows when the network cannot deliver packets at the rate they are injected. Packets are immediately removed upon reaching their destination endnode. The number of VCs available is 4. Each routing algorithm, TOR, DOR or $Up^*/Down^*$, evenly distributes the paths among the available VCs in order to achieve a balanced load.

A transparent synthetic workload model is applied. For the packet injection rate a normal approximation of the Poisson distribution is used. We study two different traffic patterns: A uniform destination address distribution, and a hotspot traffic pattern where 80% of the packets are destined for a randomly selected hotspot node whereas the remaining 20% of the packets are uniformly distributed.

In order to ensure relevant load levels for the experiments, we initially identified the saturation point (the load level where the transmission queues of the endnodes start to overflow) for each of the three routing algorithms.

For uniform traffic, $Up^*/Down^*$ has the lowest saturation point (Sat_{min}) of the three routing algorithms, whereas the highest saturation point (Sat_{max}) is achieved for TOR and DOR for the mesh topology, and for DOR for the torus

⁶ Among the upper ten supercomputers in the Top500 list [26], both mesh and torus topologies are represented.

topology. We selected three load levels of focus, where the lowest, medium and highest load levels correspond to 90% of Sat_{min} , the center between Sat_{min} and Sat_{max} , and 110% of Sat_{max} , respectively.

For hotspot traffic, the saturation point (Sat) is the same for all three routing algorithms (as the saturation point is mainly decided by the congestion that results from 80% of the traffic being directed towards one of the endnodes). Two load levels were in focus, where the lowest and highest level correspond to 80% and 110%, respectively, of Sat .

In order to remove initial transients, data collection is not started for an experiment until 50000 cycles have been run (time is measured in cycles – an abstract time unit)⁷. Data are collected for 100000 cycles. A random link (port) fault occurs after 6000 cycles and a random switch fault occurs after 28000 cycles. In each case a reconfiguration process is triggered.

Using OR, a traffic source (endnode) injects a token to indicate that no packets routed according to R_{old} will follow. The change-over from one routing function to another could be fully synchronized if all traffic sources performed the change simultaneously. It is well known that, due to such factors as clock skew or reception of routing or control information at different times, such synchronization is hard to achieve. In order to compare RecTOR and OR for different degrees of synchronization, we use source routing and let the endnodes perform the change-over from R_{old} to R_{new} as follows. When all endnodes have been notified to initialize reconfiguration, each endnode draws its change-over time, t_{change} , from a normal distribution with a mean of 500 cycles and where the standard deviation is a simulation parameter, $change_{std}$. An endnode starts a timer according to t_{change} and continues injecting packets_{old} until the timer expires, then, if OR is used, injects the token, and thereafter injects packets_{new}. The higher $change_{std}$ is, the more unsynchronized the change of routing function becomes. In these experiments, $change_{std}$ assumes the values 0 and 100, where the former value represents the fully synchronized case.

We consider the metrics Thr_t and Lat_t which result from the division of the data collection period into 500 time intervals, each with a duration of 200 cycles. For a time interval int , Thr_t is the number of packets that are generated by any endnode in int and that subsequently reach their destination endnode. Lat_t for int is the average latency of all packets that are generated by any endnode in int and that subsequently reach their destination endnode. The latency for a single packet is the time that elapses from when the packet is generated and injected into a transmission queue in the source endnode until the packet is received by the destination endnode. For each int the values for Thr_t and Lat_t are plotted in the middle of the interval, whereas in the same plots the start and end times of the reconfiguration period are plotted without regard to interval borders. For OR the reconfiguration starts when the first token is injected and ends when the last token is received by an endnode. For RecTOR the reconfiguration starts when the first endnode starts using the new routing function and ends when the last packet that belonged to the old routing function are removed from the network.

⁷ E.g. a link speed of 10 Gbps gives a simulator cycle length of 102.4 ns.

The values presented are the mean values that result from 30 repetitions of each experiment. Each repetition is initialized by a different seed and applies a unique link (port) fault, switch fault, and hotspot node (in the case of hotspot traffic).

6 Results

Due to space limitations we could not display the plots from all our experiments. Therefore, a set of representative plots was selected and included.

For the uniform traffic pattern, Figure 1 compares RecTOR with OR_{TOR} and OR_{DOR/UD} for a 8×8 mesh under low traffic load and a 8×8 torus under medium traffic load. The vertical lines depict the start and end times of reconfiguration for each of the three methods (some of the lines are plotted on top of each other). The reduction of Thr_t seen at times 6000 and 28000 for all three methods is due to packet loss in the faulty link and switch, respectively.

Figures 1(a) and 1(b) show that, after the first reconfiguration, RecTOR and OR_{TOR} achieve a significantly higher Thr_t than OR_{DOR/UD} does. Likewise, Figures 1(c) and 1(d) show that, after the first reconfiguration, RecTOR and OR_{TOR} achieve a significantly lower Lat_t than OR_{DOR/UD} does. The low load applied in Figures 1(a) and 1(c) is below saturation for Up*/Down* in the fault-free case. Nevertheless, the fault of only a single link causes a significant performance degradation for Up*/Down* when compared to TOR.

Using DOR for a fault-free mesh or torus, and then using Up*/Down* to calculate new routes as a fault occurs, is a relatively common approach.⁸ However, at least for a traffic pattern that resembles uniform, Figure 1 clearly demonstrates the drawbacks of such an approach. RecTOR assumes TOR, which not only achieves better performance than Up*/Down* after a fault has occurred, but also matches the performance of DOR in the fault-free case. Furthermore, using only one routing algorithm simplifies the implementation.

OR often causes decreased throughput and increased latency during the reconfiguration as a number of packets_{new} are held back by the token propagation procedure. For OR_{TOR} the characteristic troughs of the Thr_t curves and crests of the Lat_t curves were hardly noticeable in Figure 1 due to the large scale of these plots. Figure 2, on the other hand, clearly shows the advantages of RecTOR over OR_{TOR} in the case of uniform traffic. OR_{DOR/UD} is not included as we have already concluded from Figure 1 that its performance is inferior to RecTOR and OR_{TOR}.

For a 16×16 mesh, Figure 2(a) shows Thr_t for medium load and Figure 2(c) shows Lat_t for low load. For a 16×16 torus, Figure 2(b) shows Thr_t for high load and Figure 2(d) shows Lat_t for medium load. All of these plots show that, for OR_{TOR}, the Thr_t and Lat_t curves have significant troughs and crests, respectively. For RecTOR, on the other hand, there are no troughs in the Thr_t curves nor crests in the Lat_t curves, as no restrictions are placed on the forwarding of packets during reconfiguration. The decreased throughput and increased latency observed for RecTOR after a fault are merely due to the reduced capacity of

⁸ Remember that DOR only works for fault-free meshes and tori.

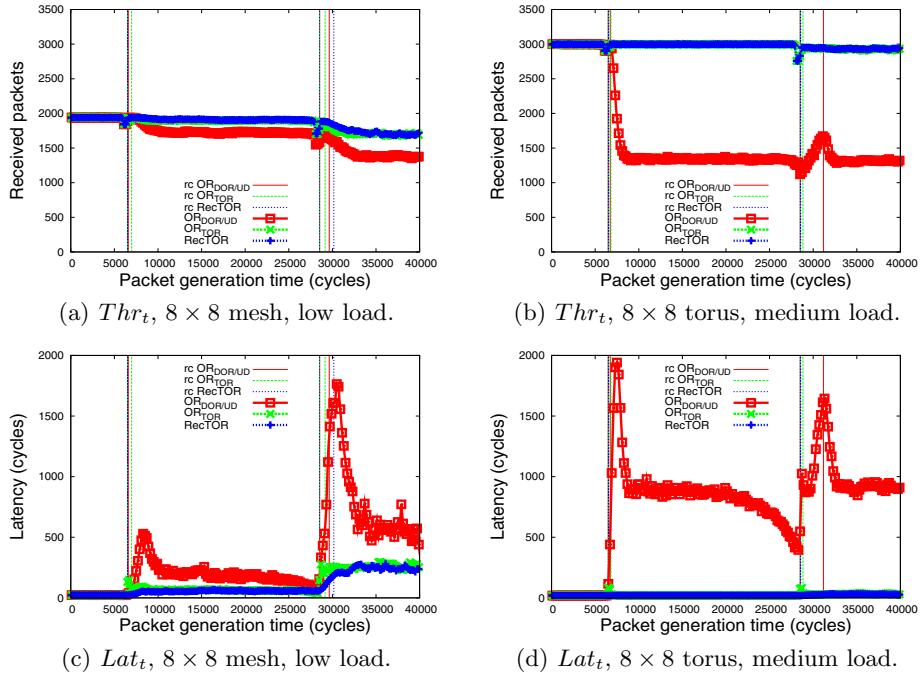


Fig. 1. $Thrt$, Lat_t , and start/end times of reconfiguration (rc) for RecTOR, OR_{TOOR} and OR_{DOR/UD} (uniform traffic, synchronized change-over to R_{new})

the network when packets can no longer be forwarded across the faulty link or switch (naturally, this effect is also observed for OR_{TOOR}).

RecTOR performs equally well in the case of a less synchronized change-over ($change_{std} = 100$) from R_{old} to R_{new} as in the case of a fully synchronized change-over ($change_{std} = 0$). Therefore, Figure 2 includes only the less synchronized case for RecTOR, whereas both the fully and less synchronized cases are included for OR_{TOOR}. Figure 2(b) shows that, for a traffic load well above saturation, the performance of OR_{TOOR} does not depend on how synchronized the change-over from R_{old} to R_{new} is. For low and medium traffic load, on the other hand, Figures 2(a), 2(c) and 2(d) demonstrate that, for OR_{TOOR}, the troughs of the $Thrt$ curves grow deeper, and the crests of the Lat_t curves grow higher as the change-over from R_{old} to R_{new} gets less synchronized.

In summary, Figure 2 shows that RecTOR provides a better network service to an application with a uniform communication pattern than OR does, and that the advantages become even more apparent if the change-over from R_{old} to R_{new} is not fully synchronized.

For the hotspot traffic pattern, Figure 3 compares RecTOR with OR_{TOOR} and OR_{DOR/UD}. Figures 3(a) and 3(b) show Lat_t for a 16×16 torus under low traffic load. Figure 3(a) shows a fully synchronized change-over from R_{old} to R_{new} , whereas Figure 3(b) shows the less synchronized change-over. For RecTOR, the

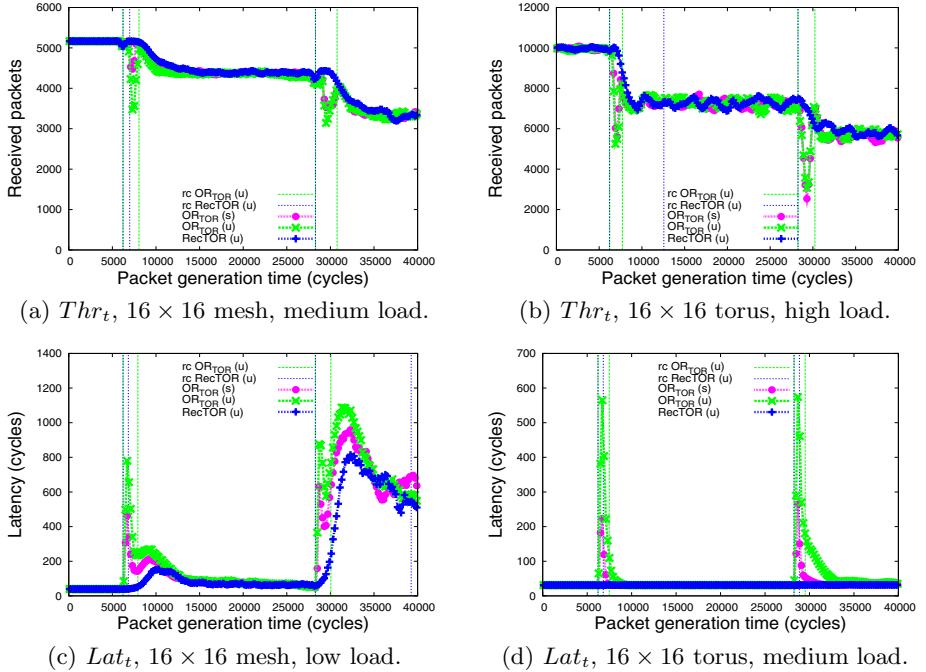


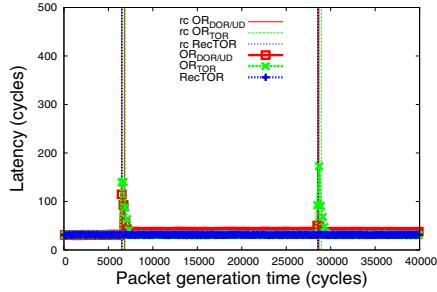
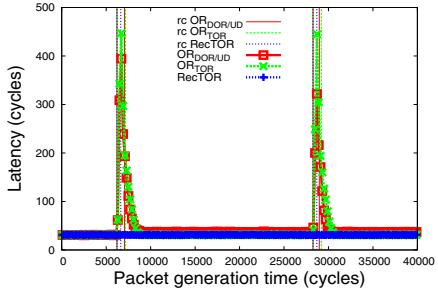
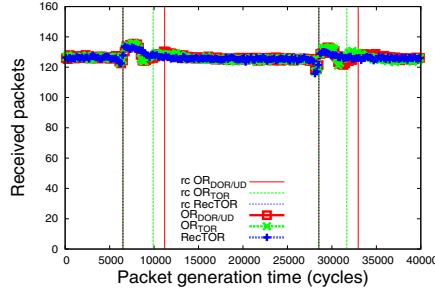
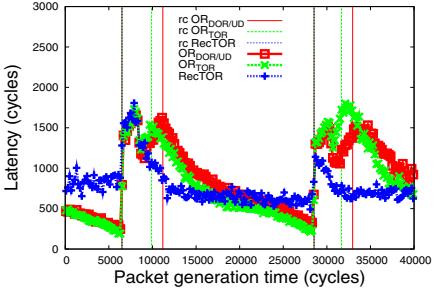
Fig. 2. $Thrt$, $Latrt$, and start/end times of reconfiguration (rc) for RecTOR and OR_{TOOR} under synchronized (s) and unsynchronized (u) change-over to R_{new} (uniform traffic)

$Latrt$ curves are smooth, without any crests, and, as we already know, the performance of RecTOR is independent of how synchronized the change-over to R_{new} is. For OR_{TOOR} and OR_{DOR/UD}, on the other hand, the crests of the $Latrt$ curves are pronounced. For both OR_{TOOR} and OR_{DOR/UD} the heights of these crests increase significantly as the change-over to R_{new} gets less synchronized. Thus, as for uniform traffic, the advantages of RecTOR over OR become even more apparent when the change-over from R_{old} to R_{new} is not fully synchronized.

Figures 3(a) and 3(b) also indicate that after the first reconfiguration, $Latrt$ for OR_{DOR/UD} is higher than for RecTOR and OR_{TOOR}. This is due to inferior performance of Up*/Down* when compared to TOR, as was also demonstrated for uniform traffic in Figure II.

For hotspot traffic, the packet throughput is limited due to 80% of the traffic being directed towards one particular node. This explains that in Figure 3(c), which shows $Thrt$ for a 8 × 8 mesh under high load, the characteristic troughs in the $Thrt$ curves for OR are barely visible.⁹ We believe that the small increase in $Thrt$ immediately after the start of each reconfiguration in Figure 3(c) is due to more packets being accepted into the network when the new routes around a

⁹ For low traffic load, the $Thrt$ curves simply resembled horizontal lines, except for the packet loss in a faulty link or switch. Therefore, none of these plots were included.

(a) Lat_t , 16×16 torus, low load, synchronized change-over to R_{new} .(b) Lat_t , 16×16 torus, low load, unsynchronized change-over to R_{new} .(c) Thr_t , 8×8 mesh, high load, synchronized change-over to R_{new} .(d) Lat_t , 8×8 mesh, high load, synchronized change-over to R_{new} .**Fig. 3.** Thr_t , Lat_t , and start/end times of reconfiguration (rc) for RecTOR, OR_{TOR} and OR_{DOR/UD} (hotspot traffic)

faulty link or switch are taken into use. Figure 3(d) shows that such an increase in Thr_t corresponds to an increase in Lat_t for all three reconfiguration methods. Figure 3(d) demonstrates that, for a load level above saturation, OR experiences more fluctuations in Lat_t than RecTOR does. For OR_{TOR} and OR_{DOR/UD} the Lat_t decreases for packets_{old} injected over a period of time before reconfiguration. This effect is caused by a number of packets_{new} being held back in the switches during the token propagation procedure (which reduces the network load, and packets_{old} thus experience reduced latency). On the other hand, some significant crests in the Lat_t curves for OR_{TOR} and OR_{DOR/UD} are also results of the token propagation procedure. Figures 3(c) and 3(d) represent an extreme scenario (a heavy hotspot pattern in combination with a workload well above saturation). However, even in this case, RecTOR supports a more stable network service than OR does.

7 Conclusion

Existing methods for reconfiguration of an interconnection network have a number of limitations, such as dependence on deadlock-avoidance mechanisms that

impose performance penalties; complicated procedures; or dependence on the Up*/Down* routing algorithm which achieves low performance. Some of the methods require complex network switches, or are only applicable for distributed routing systems.

This paper presents RecTOR, a new dynamic reconfiguration method, which is useful both for source and distributed routing systems, and which does not require complex network switches. RecTOR is based on a simple principle that, while ensuring deadlock-freedom, allows packets routed according to an old and a new routing function to coexist in the network without restrictions. Thus, unlike e.g. OR, RecTOR does not cause degraded performance during the reconfiguration.

Our performance evaluation shows that, during the reconfiguration, RecTOR supports a better network service than OR does, both to applications with uniform and hotspot communication patterns. Complete synchronization of the change-over from an old to a new routing function is hard to achieve (due to such factors as clock skew or reception of routing or control information at different times). The simulation results show that the advantages of RecTOR over OR become even more evident as the change-over gets less synchronized. Furthermore, our results demonstrate the limitations of the common approach that implies using DOR in a fault-free mesh or torus, and then using Up*/Down* routing if a fault occurs. Using RecTOR, only one routing algorithm is needed – TOR, a topology agnostic routing algorithm that not only outperforms Up*/Down*, but also matches the performance of DOR.

Currently, RecTOR appears as the most efficient reconfiguration method for systems that accept out-of-order packet delivery and have virtual channels available for the routing function.

References

1. Duato, J., Yalamanchili, S., Ni, L.: *Interconnection Networks: An Engineering Approach*. Morgan Kaufmann Publishers, San Francisco (2003)
2. Duato, J., Lysne, O., Pang, R., Pinkston, T.M.: Part I: A theory for deadlock-free dynamic network reconfiguration. *IEEE Trans. Parallel and Distributed Systems* 16(5), 412–427 (2005)
3. Schroeder, M.D., et al.: Autonet: A high-speed, self-configuring local area network using point-to-point links. SRC Research Report 59, Digital Equipment Corporation (1990)
4. Rodeheffer, T.L., Schroeder, M.D.: Automatic reconfiguration in Autonet. In: 13th ACM Symp. Operating Systems Principles, pp. 183–197 (1991)
5. Boden, N.J., et al.: Myrinet: A gigabit-per-second local area network. *IEEE Micro.* 15(1), 29–36 (1995)
6. Teodosiu, D., et al.: Hardware fault containment in scalable shared-memory multiprocessors. *SIGARCH Computer Architecture News* 25(2), 73–84 (1997)
7. Lysne, O., Duato, J.: Fast dynamic reconfiguration in irregular networks. In: Int'l. Conf. Parallel Processing, pp. 449–458 (2000)
8. Casado, R., Bermúdez, A., Duato, J., Quiles, F.J., Sánchez, J.L.: A protocol for deadlock-free dynamic reconfiguration in high-speed local area networks. *IEEE Trans. Parallel and Distributed Systems* 12(2), 115–132 (2001)

9. Natchev, N., Avresky, D., Shurbanov, V.: Dynamic reconfiguration in high-speed computer clusters. In: 3rd Int'l. Conf. Cluster Computing, pp. 380–387 (2001)
10. Pinkston, T.M., Pang, R., Duato, J.: Deadlock-free dynamic reconfiguration schemes for increased network dependability. *IEEE Trans. Parallel and Distributed Systems* 14(8), 780–794 (2003)
11. Lysne, O., Pinkston, T.M., Duato, J.: Part II: A methodology for developing deadlock-free dynamic network reconfiguration processes. *IEEE Trans. Parallel and Distributed Systems* 16(5), 428–443 (2005)
12. Avresky, D., Natchev, N.: Dynamic reconfiguration in computer clusters with irregular topologies in the presence of multiple node and link failures. *IEEE Trans. Computers* 54(5), 603–615 (2005)
13. Acosta, J.R., Avresky, D.R.: Dynamic network reconfiguration in presence of multiple node and link failures using autonomous agents. In: 2005 Int'l. Conf. Collaborative Computing: Networking, Applications and Worksharing (2005)
14. Acosta, J.R., Avresky, D.R.: Intelligent dynamic network reconfiguration. In: Int'l. Parallel and Distributed Processing Symp. (2007)
15. Robles-Gómez, A., Bermúdez, A., Casado, R., Solheim, Å.G.: Deadlock-free dynamic network reconfiguration based on close Up*/Down* graphs. In: Luque, E., Margalef, T., Benítez, D. (eds.) *Euro-Par 2008. LNCS*, vol. 5168, pp. 940–949. Springer, Heidelberg (2008)
16. InfiniBand Trade Association: InfiniBand Architecture Specification v. 1.2.1 (2007), <http://www.infinibandta.org/specs>
17. Zafar, B., Pinkston, T.M., Bermúdez, A., Duato, J.: Deadlock-free dynamic reconfiguration over InfiniBand networks. *Int'l. Jrnl. Parallel, Emergent and Distributed Systems* 19(2), 127–143 (2004)
18. Bermúdez, A., Casado, R., Quiles, F.J., Duato, J.: Handling topology changes in InfiniBand. *IEEE Trans. Parallel and Distributed Systems* 18(2), 172–185 (2007)
19. Lysne, O., et al.: Simple deadlock-free dynamic network reconfiguration. In: 11th Int'l. Conf. High Performance Computing, pp. 504–515 (2004)
20. Lysne, O., et al.: An efficient and deadlock-free network reconfiguration protocol. *IEEE Trans. Computers* 57(6), 762–779 (2008)
21. Solheim, Å.G., et al.: Efficient and deadlock-free reconfiguration for source routed networks. In: 9th Worksh. Communication Architecture for Clusters (2009)
22. Sancho, J.C., Robles, A., Flích, J., López, P., Duato, J.: Effective methodology for deadlock-free minimal routing in InfiniBand networks. In: Int'l. Conf. Parallel Processing, pp. 409–418 (2002)
23. Dally, W.J., Seitz, C.L.: Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Trans. Computers* 36(5), 547–553 (1987)
24. Francois, P., Bonaventure, O.: Avoiding transient loops during IGP convergence in IP networks. In: 24th IEEE INFOCOM, vol. 1, pp. 237–247 (2005)
25. Tyan, H.-Y.: Design, Realization and Evaluation of a Component-Based Compositional Software Architecture for Network Simulation. PhD thesis, Ohio State University (2002)
26. Top 500 project: Top 500 Supercomputer Sites (November 2008), <http://top500.org>

NIC-Assisted Cache-Efficient Receive Stack for Message Passing over Ethernet

Brice Goglin

INRIA Bordeaux – France

Brice.Goglin@inria.fr

Abstract. High-speed networking in clusters usually relies on advanced hardware features in the NICs, such as zero-copy. Open-MX is a high-performance message passing stack designed for regular Ethernet hardware without such capabilities.

We present the addition of multiqueue support in the Open-MX receive stack so that all incoming packets for the same process are treated on the same core. We then introduce the idea of binding the target end process near its dedicated receive queue. This model leads to a more cache-efficient receive stack for Open-MX. It also proves that very simple and stateless hardware features may have a significant impact on message passing performance over Ethernet.

The implementation of this model in a firmware reveals that it may not be as efficient as some manually tuned micro-benchmarks. But our multiqueue receive stack generally performs better than the original single queue stack, especially on large communication patterns where multiple processes are involved and manual binding is difficult.

1 Introduction

The emergence of 10 gigabit/s ETHERNET hardware raised the questions of when and how the long-awaited convergence with high-speed networks will become a reality. ETHERNET now appears as an interesting networking layer within local area networks for various protocols such as FCoE [4]. Meanwhile, several network vendors that previously focussed on high-performance computing added interoperability with ETHERNET to their hardware, such as MELLANOX CONNECTX [3] or MYRICOM MYRI-10G [11]. However, the gap between these advanced NICs and regular ETHERNET NICs remains very large.

OPEN-MX [6] is a message passing stack implemented on top of the ETHERNET software layer of the LINUX kernel. It aims at providing high-performance communication over any generic ETHERNET hardware using the wire specifications and the application programming interface of *Myrinet Express* [12]. While being compatible with any legacy ETHERNET NICs, OPEN-MX suffers from limited hardware features.

We propose to improve the cache-efficiency of the receive side by extending the hardware IP multiqueue support to filter OPEN-MX packets as well. Such a stateless feature requires very little computing power and software support

compared to the existing complex and statefull features such as zero-copy or TOE (*TCP Offload Engine*). Parallelizing the stack is known to be important on modern machines [16]. We are looking at it in the context of binding the whole packet processing to the same core, from the bottom interrupt handler up to the application.

The remaining of this paper is organized as follows. We present OPEN-MX, its possible cache-inefficiency problems, and our objectives and motivations in Section 2. Section 3 describes our proposal to combine the multiqueue extension in the MYRI-10G firmware and its corresponding support in OPEN-MX so as to we build an automatic binding facility for both the receive handler in the driver and the target application. Section 4 presents a performance evaluation which shows that our model achieves satisfying performance for micro-benchmarks, reduces the overall cache miss rate, and improves large communication patterns. Before concluding, related works are discussed in Section 5.

2 Background and Motivations

In this section, we briefly describe the OPEN-MX stack before discussing how the cache is involved on the receive side. We then present our motivation to add some OPEN-MX specific support in the NIC and detail our objectives with this implementation.

2.1 Design of the Open-MX Stack

The OPEN-MX stack aims at providing high-performance message passing over any generic ETHERNET hardware. It exposes the *Myrinet Express API* (MX) to user-space applications. Many existing middleware projects such as OPEN MPI [5] or PVFS2 [13] run successfully unmodified on top of it.

OPEN-MX was first designed as an emulated MX firmware in a LINUX kernel module [6]. This way, legacy applications built for MX benefit from the same abilities without needing the MYRICOM hardware or the native MX software stack (see Figure 1). However, the features that are usually implemented in the hardware of high-speed networks are obviously prone to performance issues when emulated in software. Indeed, portability to any ETHERNET hardware requires the use of a common very simple low-level programming interface to access drivers and NICs.

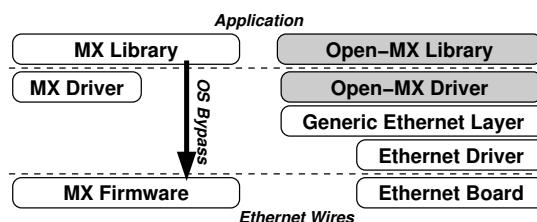


Fig. 1. Design of the native MX and generic Open-MX software stacks

2.2 Cache Effects in the Open-MX Receive Stack

Reducing cache effects in the OPEN-MX stack requires to ensure that data structures are not used concurrently by multiple cores. Since the send side is mostly driven by the application, the whole send stack is executed by the same core. The receive side is however much more complex. OPEN-MX processes incoming packets in its *Receive handler* which is invoked when the ETHERNET NIC raises an interrupt. The receive handler first acquires the descriptor of the communication channel (*endpoint*). Then, if the packet is part of a *eager* message (≤ 32 kB), the data and corresponding event are written into a ring shared with the user-space library. Finally, the library will copy the data back to the application buffers (see Figure 2).

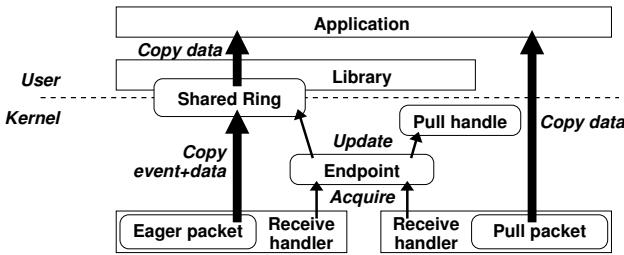


Fig. 2. Summary of resource accesses and data transfers from the Open-MX receive handler in the driver up to the application buffers

If the packet is part of a large message (after a *rendezvous*), the corresponding *Pull* handle is acquired and updated. Then, the data is copied into the associated receive buffer (Figure 2). An event is raised at the user-space level only when the last packet is received. This copy may be offloaded to INTEL *I/O Acceleration Technology* (I/OAT) DMA engine hardware if available [7].

The current OPEN-MX receive stack will most of the times receive IRQ (*Interruption ReQuest*) on all cores since the hardware chipset usually distributes them in a round-robin manner (see Figure 3(a)). It causes many cache-line bounces between cores that access the same resources. It explains why processing all packets for the same endpoint on the same core will improve the cache-efficiency. Indeed, there would be no more concurrent accesses to the endpoint structure or shared ring in the driver. Additionally, all eager packets will also benefit from having the user-space library run on the same core since a shared ring is involved.

Large message (*Pull* packets) will also benefit from having their handle accessed by a single core. But, it is actually guaranteed by the fact that each handle is used by a single endpoint. Moreover, running the application on the same core will reduce cache effects when accessing the received data (except if the copy was offloaded to the I/OAT hardware which bypasses the cache).

2.3 Objectives and Motivations

A simple way to avoid concurrent accesses in the driver is to bind the interrupt to a single core. However, the chosen core will be overloaded, causing an availability imbalance between cores. Moreover, all processes running on other cores will suffer from cache-line bounces in their shared ring since they would compete with the chosen core. In the end, this solution may only be interesting for benchmarking purposes with a single process per node (see Section 4).

High-speed networks do not suffer from such cache-related problems since events and data are directly deposited in the user-space application context. It is one of the important features that legacy ETHERNET hardware lacks. The host only processes incoming packets when the NIC raises an interrupt. Fortunately, some ETHERNET-specific hardware optimizations have been developed in the context of IP networks. The study of cache-efficiency lead to the emergence of hardware multiqueue support. These NICs have the ability to split the incoming packet flow into several queues [17] with different interrupts. By filtering packets depending on their IP connection and binding each queue to a single core, it is possible to ensure that all packets of a connection will be processed by the same core. It prevents many cache-line bounces in the host receive stack.

We propose in this article to study the addition of OPEN-MX-aware multiqueue support. We expect to improve the cache-efficiency of our receive stack by guaranteeing that all packets going to the same endpoint are processed on the same core. To improve performance even more, we then propose to bind the target user-process to the core where the endpoint queue is processed. It will make the whole OPEN-MX receive stack much more cache-friendly. This idea goes further than existing IP implementations where the cache-efficiency does not go up to the application.

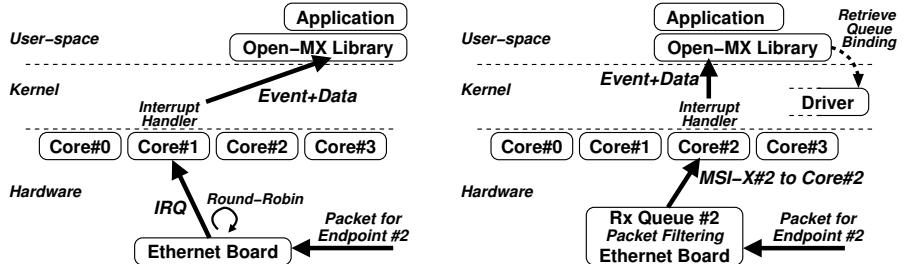
3 Design of a Cache-Friendly Open-MX Receive Stack

We now detail our design and implementation of a cache-friendly receive stack in OPEN-MX thanks to the addition of dedicated multiqueue support in the NIC and the corresponding process binding facility.

3.1 Open-MX-Aware Multiqueue Ethernet Support

Hardware multiqueue support is based on the driver allocating one MSI-X interrupt vector (similar to an IRQ line) and one ring per receive queue. Then, for each incoming packet, the NIC decides which receive queue should be used [17]. The IP traffic is dispatched into multiple queues by hashing each connection into a queue index.

The OPEN-MX multiqueue support is actually very simple because hashing its packets is easy. Indeed, the same communication channel (*endpoint*) is used to communicate with many peers, so only the local endpoint identifier has to be hashed. Therefore, the NIC only has to convert the 8-bit destination endpoint



(a) Round-Robin Single-Interrupt: the interrupt goes to any core while the application may run somewhere else.

(b) OPEN-MX-aware Multiqueue: the NIC raises the MSI-X interrupt corresponding to the core where the application runs.

Fig. 3. Path from the NIC interrupt up to the application receiving the event and data

identifier into a queue index. It is much more simple than hashing IP traffic where many connection parameters (source and destination, port and address) have to be involved in the hash function. This model is summarized in Figure 3(b). The next step towards a cache-friendly receive stack is to bind each process to the core which handles the receive queue of its endpoint.

3.2 Multiqueue-Aware Process Binding

Now that the receive handler is guaranteed to run on the same core for all packets of the same endpoint, we discuss how to have the application run there as well. One solution would be to bind the receive queue to the current core when an endpoint is open. However, the binding of all queues has to be managed globally so that the multiqueue IP is not disturbed by a load imbalance between cores.

We have chosen the opposite solution: keep receive queues bound as usual (one queue per core) and make OPEN-MX applications migrate on the right core when opening an endpoint. Since most high-performance computing applications place one process per core, and since most MPI implementations use a single endpoint per process, we expect each core to be used by a single endpoint. In the end, each receive queue will actually be used by a single endpoint as well. It makes the whole model very simple.

3.3 Implementation

We implemented this model in the OPEN-MX stack with MYRICOM MYRI-10G NICs as an experimentation hardware. We have chosen this board because it is one of the few NICs with multiqueue receive support. It also enables comparisons with the MX stack which may run on the same hardware (with a different firmware and software stack that was designed for MPI).

We implemented the proposed modification in the `myri10ge` firmware by adding our specific packet hashing. It decodes native OPEN-MX packet headers

to find out the destination endpoint number. Once the ETHERNET driver has been setup with one receive queue per core as usual, each endpoint packet flow is sent to a single core.

Meanwhile, we added to the `myri10ge` driver a routine that returns the MSI-X interrupt vector that will be used for each OPEN-MX endpoint. When OPEN-MX attaches an interface whose driver exports such a routine, it gathers all interrupt affinities (the binding of the receive queues). Then, it provides the OPEN-MX user-space library with binding hints when it opens an endpoint. Applications are thus automatically migrated onto the core that will process their packets. It makes the whole stack more cache-friendly, as described on Figure 3(b).

4 Performance Evaluation

We now present a performance evaluation of our model. After describing our experimentation platform, we will detail micro-benchmarks and application-level performance.

4.1 Experimentation Platform

Our experimentation platform is composed of 2 machines with 2 INTEL XEON E5345 quad-core *Clovertown* processors (2.33 GHz). These processors are based on 2 dual-core subchips with a shared L2 cache as described in Figure 4. It implies 4 possible processor/interrupt bindings : on the same core (SC), on a core sharing a cache (SS), on another core of the same processor (SP), and on another processor (OP).

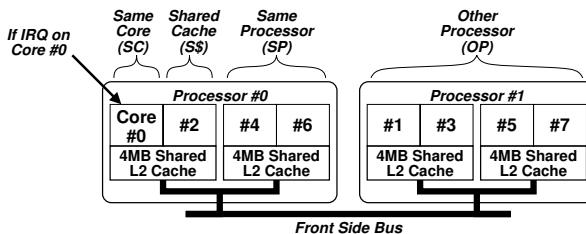


Fig. 4. Processors and caches in the experimentation platform (OS-numbered)

These machines are connected with MYRI-10G interfaces running in ETHERNET mode with our modified `myri10ge` firmware and driver. We use OPEN MPI 1.2.6 [5] on top of OPEN-MX 0.9.2 with LINUX kernel 2.6.26.

4.2 Impact of Binding on Micro-Benchmarks

Table 10 presents the latency and throughput of *Intel MPI Benchmark* [10] PINGPONG depending on the process and interrupt binding. It first shows that

Table 1. IMB PINGPONG performance depending on process and IRQ binding

Metric	Binding	SC	S\$	SP	OP
0 byte latency	Round-Robin Single IRQ	$\simeq 11.4 \mu s$			
	Single IRQ on core #0	10.96	9.34	10.32	10.25
	Multiqueue	11.71	10.10	11.09	11.25
4 MB throughput	Round-Robin Single IRQ	$\simeq 646 \text{ MiB/s}$			
	Single IRQ on core #0	719	723	721	714
	Multiqueue	703	707	706	697
4 MB throughput (I/OAT)	Round-Robin Single IRQ	$\simeq 905 \text{ MiB/s}$			
	Single IRQ on core #0	1056	1059	1048	1026
	Multiqueue	955	965	948	938

the original model (with a single interrupt dispatched to all cores in a round-robin manner) is slower than any other model, due to cache-line bounces. When binding the single interrupt to a single core, the best performance is achieved when the process and interrupt handler share a cache. Indeed, this case reduces the overall latency thanks to cache hits in the receive stack, while it prevents the user-space library and interrupt handler from competing for the same core.

Multiqueue support achieves satisfying performance, but remains a bit slower than optimally bound single interrupt. It is related to the multiqueue implementation requiring more work in the NIC than the single interrupt firmware.

4.3 Idle Core Avoidance

The above results assumed that one process was running on each core even if only two of them were actually involved in the MPI communication. This setup has the advantage of keeping all cores busy. However, it may be far from the behavior of real applications where for instance disk I/O may put some processes to sleep and cause some cores to become idle. If an interrupt is raised onto such an idle core, it will have to wakeup before processing the packet. On modern processors, this wakeup overhead is several microseconds, causing the overall latency to increase significantly.

To study this problem, we ran the previous experiment with only one communicating process per node, which means 7 out of 8 cores are idle. When interrupts are not bound to the right core, it increases the latency from 11 up to 15-20 μs and reduces the throughput by roughly 20 %.

This result is another justification of our idea to bind the process to the core that runs its receive queue. Indeed, if a MPI application is waiting for a message, the MPI implementation will usually busy poll the network. Its core will thus not enter any sleeping state. By binding the receive queue interrupt and the application to the same core, we guarantee that this busy polling core will be the one processing the incoming packet in the driver. It will be able to process it immediately, causing the observed latency to be much lower. All other cores that may be sleeping during

disk I/O will not be disturbed by packet processing for unrelated endpoints. This result may even reduce the overall power consumption of the machine.

4.4 Cache Misses

Table 2 presents the percentage of cache misses observed with PAPI [2] during a ping-pong depending on interrupt and process binding. Only L2 cache accesses are presented since the impact on L1 accesses appears to be lower.

Table 2. L2 cache misses (Kernel+User) during a ping-pong

Length	Round-Robin IRQ	IRQ on S\$	IRQ on SC
0 B	29.80 % \pm 13.79 %	29.70 % \pm 8.34 %	11.47 % \pm 0.13 %
128 B	26.80 % \pm 17.90 %	25.06 % \pm 23.05 %	14.15 % \pm 0.51 %
32 kB	28.77 % \pm 17.71 %	23.17 % \pm 29.32 %	24.49 % \pm 22.56 %
1 MB (I/OAT)	27.7 % \pm 6.28 %	36.9 % \pm 7.97 %	25.20 % \pm 5.96 %

The table first shows that the cache miss rate is dramatically reduced for small messages thanks to our multiqueue support. Running the receive handler (the kernel part of the stack) always on the same core divides cache misses in the kernel by 2. Binding the target application (the user part of the stack) to the same core reduces user-space cache misses by a factor of up to 100.

Cache misses are not improved for 32 kB message communication. We expect this behavior to be related to many copies being involved on the receive path. It causes too many cache pollution, which prevents our cache efficiency from being useful.

Very large messages with I/OAT copy offload do not involve any data copy in the receive path. Cache misses are thus mostly related to concurrent accesses to the endpoint and pull handles in the driver. We observe a slightly decreased cache miss rate thanks to proper binding. But the overall rate remains high, likely because it involves some code-paths outside of the OPEN-MX receive stack (*rendezvous* handshake, send stack, ...) which are expensive for large messages.

4.5 Collective Communication

After proving that our design improves cache-efficiency without strongly disturbing micro-benchmark performance we now focus on complex communication patterns by first looking at collective operations. We ran IMB ALLTOALL between our nodes with one process per core. Figure 5 presents the execution time compared to the native MX stack, depending on interrupt and receive queue binding. It shows that using a single receive queue results in worse performance than our multiqueue support. As expected, binding this single interrupt to a single core decreases the performance as soon as the message size increases since the load on this core becomes the limiting factor.

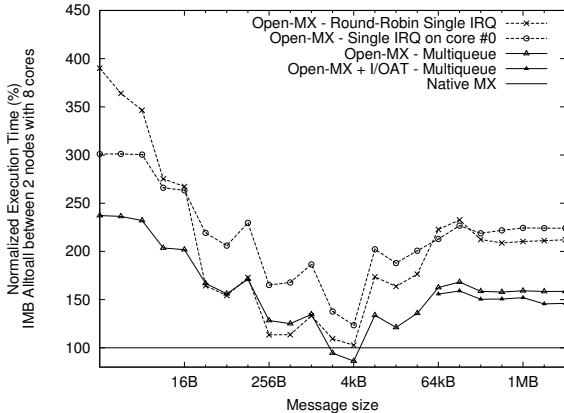


Fig. 5. IMB ALLTOALL relative execution time depending on interrupt binding and multiqueue support, compared with the native MX stack

When multiqueue support is enabled, the overall ALLTOALL performance is on average 1.3 better. It now reaches less than 150 % of the native MX stack execution time for very large messages when I/OAT copy offload is enabled. Moreover, our implementation is even able to outperform MX near 4 kB message sizes¹.

This result reveals that our implementation achieves its biggest improvement when the communication pattern becomes larger and more complex (collective operation with many local processes). We think it is caused by such patterns requiring more data transfer within the host and thus making cache-efficiency more important.

4.6 Application-Level Performance

Table 3 presents the execution time of some *NAS Parallel Benchmarks* [11] between our two 8-core hosts. Most programs show a few percents performance improvement thanks to our work. This impact is limited by the fact that these applications are not highly communication intensive. IS (which performs many large message communications) shows an impressive speedup (8.5 for class B, 2.6 for class C). Thanks to our multiqueue support, IS is now even faster on OPEN-MX than on MX. We feel that such a huge speedup cannot be only related to the efficiency of our new implementation. It is probably also caused by poor performance of the initial single-queue model for some reason.

It is again worth noticing that using a single interrupt bound to a single core sometimes decreases performance. As explained earlier, this configuration should only be preferred for micro-benchmarking with very few processes per node.

¹ OPEN-MX mimics MX behavior near 4 kB. This message size is a good compromise between smaller sizes (where the big ETHERNET latency matters) and larger messages (where intensive memory copies may limit performance).

Table 3. NAS Parallel Benchmark execution time and improvement

	Single IRQ Round-Robin	Single IRQ on Single core	Multiqueue	Performance Improvement	MX
cg.B.16	34.62 s	34.32 s	33.68 s	+2.8 %	32.23 s
mg.B.16	4.14 s	4.19 s	4.02 s	+2.9 %	3.93 s
ft.B.16	22.80 s	23.06 s	21.34 s	+6.8 %	19.61 s
is.B.16	11.84 s	10.83 s	1.25 s	$\times 8.5$	1.33 s
is.C.16	14.69 s	14.17 s	5.62 s	$\times 2.6$	6.30 s

5 Related Works

High-performance communication in clusters heavily relies on specific features in the networking hardware. However, they are usually very different from the multiqueue support that we presented in this paper. The most famous hardware feature for HPC remains zero-copy support. It has also been added to some ETHERNET-based message passing stacks, for instance by relying on RDMA-enabled hardware and drivers, such as iWARP [14]. This strategy achieves a high throughput for large messages. But it requires complex modifications of the operating system (since the application must be able to provide receive buffers to the NIC) and of the NIC (which decides which buffer should be used when a new packet arrives).

Nevertheless, several other important features are still only available in HPC hardware, for instance application-directed polling for incoming packets. Indeed, high-speed network NICs have the ability to deposit events in user-space buffers where the application may poll. This innovation helps reducing the overall communication latency, but once again it requires complex hardware support. Regular ETHERNET hardware does not provide such features since it relies on a interrupt-driven model. The host processes incoming packets only when the NIC raises an interrupt. It prevents applications from polling, and implies poor cache-efficiency unless proper binding is used. Our implementation solves this problem by binding the whole receive stack properly.

Several ETHERNET-specific hardware optimizations are widely used, but they were not designed for HPC. Advanced NICs now enable the offload of TCP fragmentation/reassembly (TSO and LRO) to decrease the packet rate in the host [8]. But this work does not apply to message based protocols such as OPEN-MX. Another famous recent innovation is multiqueue support [17]. This packet filtering facility in the NIC enables interesting receive performance improvement for IP thanks to better cache-efficiency in the host. The emerging area where multiqueue support is greatly appreciated is virtualization since each queue may be bound to a virtual machine, causing packet flows to different VMs to be processed independently [15]. We adapted this hardware multiqueue support to HPC and extended it further by adding the binding of the corresponding target application. This last idea is, to the best of our knowledge, not used by any popular message passing stack such as OPEN MPI [5]. They usually just bind a single process per core, without looking at its affinity for the underlying hardware.

6 Conclusion and Perspectives

The announced convergence between high-speed networks such as INFINIBAND and ETHERNET raises the question of which specific hardware features will become legacy. While HPC networking relies on complex hardware features such as zero-copy, ETHERNET remains simple. The OPEN-MX message passing stack achieves interesting performance on top of it without requiring advanced features in the networking hardware.

This paper presents a study of the cache-efficiency of the OPEN-MX receive stack. We looked at the binding of interrupt processing in the driver and of the library in user-space. We proposed the extension of the existing IP hardware multiqueue support which assigns a single core to each connection. It prevents shared data structures from being concurrently accessed by multiple cores. OPEN-MX specific packet hashing has been added into the official firmware of MYRI-10G boards so as to associate a single receive queue with each communication channel. Secondly, we further extended the model by enabling the automatic binding of the target end application to the same core. Therefore, there are fewer cache-line bounces between cores from the interrupt handler up to the target application.

Performance evaluations first shows that the usual single-interrupt based model may achieve very good performance when using a single task and binding it so that it shares a cache with the interrupt handler. However, as soon as multiple processes and complex communication patterns are involved, the performance of this model suffers, especially from load imbalance between the cores. Using a single-interrupt scattered to all cores in a round-robin manner distributes the load but it shows limited performance due to many cache misses.

Our proposed multiqueue implementation distributes the load as well. It also offers satisfying performance for simple benchmarks. Moreover, binding the application near its receive queue further improves the overall performance thanks to fewer cache misses occurring on the receive path and thanks to the target core being ready to process incoming packets. Communication intensive patterns reveal a large improvement since the impact of cache pollution is larger when all cores and caches are busy. We observe more than 30 % of improvement for ALLTOALL operations, while the execution time of the communication intensive NAS parallel benchmark IS is reduced by a factor of up to 8.

These results prove that very simple hardware features enable significant performance improvement. Indeed, multiqueue support is becoming a standard feature that many NIC now implement. Our implementation is *Stateless* and does not require any intrusive modification of the NIC or host, contrary to usual HPC innovations. We now plan to further study hardware assistance in the context of message passing over ETHERNET. We will first look at multiqueue support on the send side. Then, designing an OPEN-MX-aware interrupt coalescing in the NIC may lead to a better compromise between latency and host load. Finally, the prefetching of incoming packets in processor caches with INTEL DCA (*Direct Cache Access*) [9] is another feature that may be combined with multiqueue

support for improving performance. Such features, as well as other stateless features than can be easily implemented in legacy NICs, open a large room for improvement of message passing over ETHERNET networks.

Acknowledgments

We would like to thank Hyong-Youb Kim, Andrew J. Gallatin, and Loïc Prylli from Myricom, Inc. for helping us when modifying the `myri10ge` firmware.

References

1. Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, D., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Schreiber, R.S., Simon, H.D., Venkatakrishnan, V., Weeratunga, S.K.: The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications* 5(3), 63–73 (Fall 1991)
2. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A Portable Programming Interface for Performance Evaluation on Modern Processors. *The International Journal of High Performance Computing Applications* 14(3), 189–204 (2000)
3. Mellanox ConnectX - 4th Generation Server & Storage Adapter Architecture, http://mellanox.com/products/connectx_architecture.php
4. FCoE (Fibre Channel over Ethernet), <http://www.fcoe.com>
5. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, concept, and design of a next generation MPI implementation. In: *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104 (2004)
6. Goglin, B.: Design and Implementation of Open-MX: High-Performance Message Passing over generic Ethernet hardware. In: *CAC 2008: Workshop on Communication Architecture for Clusters*, held in conjunction with IPDPS 2008, Miami, FL, April 2008. IEEE Computer Society Press, Los Alamitos (2008)
7. Goglin, B.: Improving Message Passing over Ethernet with I/OAT Copy Offload in Open-MX. In: *Proceedings of the IEEE International Conference on Cluster Computing*, Tsukuba, Japan, September 2008, pp. 223–231. IEEE Computer Society Press, Los Alamitos (2008)
8. Grossman, L.: Large Receive Offload Implementation in Neterion 10GbE Ethernet Driver. In: *Proceedings of the Linux Symposium (OLS 2005)*, Ottawa, Canada, July 2005, pp. 195–200 (2005)
9. Huggahalli, R., Iyer, R., Tetrick, S.: Direct Cache Access for High Bandwidth Network I/O. *SIGARCH Computer Architecture News* 33(2), 50–59 (2005)
10. Intel MPI Benchmarks, <http://www.intel.com/cd/software/products/asmo-na/eng/cluster/mpi/219847.htm>
11. Myricom Myri-10G, <http://myri.com/Myri-10G/>
12. Myricom, Inc. Myrinet Express (MX): A High Performance, Low-Level, Message-Passing Interface for Myrinet (2006), <http://www.myri.com/scs/MX/doc/mx.pdf>
13. The Parallel Virtual File System, version 2, <http://www.pvfs.org>

14. Mohammad, J.R., Afsahi, A.: 10-Gigabit iWARP Ethernet: Comparative Performance Analysis with Infiniband and Myrinet-10G. In: Proceedings of the International Workshop on Communication Architecture for Clusters (CAC), held in conjunction with IPDPS 2007, Long Beach, CA, March 2007, p. 234 (2007)
15. Santos, J.R., Turner, Y., Janakiraman, G(J.), Pratt, I.: Bridging the Gap between Software and Hardware Techniques for I/O Virtualization. In: Proceedings of USENIX 2008 Annual Technical Conference, Boston, MA, June 2008, pp. 29–42 (2008)
16. Willmann, P., Rixner, S., Cox, A.L.: An Evaluation of Network Stack Parallelization Strategies in Modern Operating Systems. In: Proceedings of the USENIX Technical Conference, Boston, MA, pp. 91–96 (2006)
17. Yi, Z., Waskiewicz, P.P.: Enabling Linux Network Support of Hardware Multiqueue Devices. In: Proceedings of the Linux Symposium (OLS 2007), Ottawa, Canada, June 2007, pp. 305–310 (2007)

A Multipath Fault-Tolerant Routing Method for High-Speed Interconnection Networks^{*}

Gonzalo Zarza, Diego Lugones, Daniel Franco, and Emilio Luque

Computer Architecture and Operating Systems Department,
University Autonoma of Barcelona, Spain
{gonzalo.zarza,diego.lugones}@caos.uab.es,
{daniel.franco,emilio.luque}@uab.es

Abstract. The intensive and continuous use of high-performance computers for executing computationally intensive applications, coupled with the large number of elements that make them up, dramatically increase the likelihood of failures during their operation.

The interconnection network is a critical part of high-performance computer systems that communicates and links together the processing units. Network faults have an extremely high impact because the occurrence of a single fault may prevent the correct finalization of applications.

This work focuses on the problem of fault tolerance for high-speed interconnection networks by designing a fault tolerant routing method. The goal is to solve a certain number of link and node failures, considering its impact, and occurrence probability. To accomplish this task we take advantage of communication path redundancy, by means of adaptive multipath routing approaches that fulfill the four phases of fault tolerance: error detection, damage confinement, error recovery, fault treatment and continuous service. Experiments show that our method allows applications to successfully finalize their execution in the presence of several number of faults, with an average performance value of 97% with respect to the fault-free scenarios.

1 Introduction

High-performance computer systems have opened a trend in modeling the modern society daily behavior and life style by means of applications and services such as wheather forecasting, geological activity studies and molecular dynamics simulations, among others. Even a simple Google search is based on high-performance computer (HPC) systems [1].

The steady increase in complexity and number of components of HPC systems leads to significantly higher failure rates. Because of this, and due to long execution times of computationally intensive applications, various computer systems show a Mean Time Between Failures (MTBF) smaller than the execution time of some of these applications [2]. This means that at least one failure will probably occur during the execution of these applications.

^{*} Supported by the MEC-Spain under contract TIN2007-64974.

Questions arise from the analysis of these situations such as: how do the failures affect these HPC systems? Are such systems able to maintain their operation and performance standards despite of failure occurrences? If they are not, What should the solution be? What are the best options to achieve fault tolerance and system service continuity?

The main purpose of fault tolerance is to enable systems to remain operational, while maintaining their performance standards as homogeneous as possible even in the presence of multiple faults. Therefore, a fault tolerant system is the one that might mask the faults occurrences by taking advantage of some kind of redundancy (hardware, software, time, etc.). In fact, redundancy is the key for supporting fault tolerance.

In this work, we focus on the fault tolerance problem for high-speed interconnection networks (HSIN) due to their primary role as the linking element of the HPC systems. As in the general system, a fault-tolerant HSIN must achieve performance standards as homogeneous as possible and, above all, allow the applications to finalize their executions even in the presence of multiple faults. There are three main approaches that could be chosen to achieve this goal: component redundancy, network reconfiguration, and fault-tolerant routing algorithms [3]. The component redundancy approach is often used in some systems but the high extra cost of the redundant spare components is an important drawback. The second approach stops the network and reconfigures the routing tables in case of a network fault in order to adapt them to the new topology after the fault. This approach is very flexible and powerful but at the expense of killing network performance. Routing algorithms designed for fault tolerance looks for alternative paths when a fault disables the original path used to communicate a pair of source-destination nodes.

Many research studies have been published in the field of fault tolerance for interconnection networks throughout the past few decades. The vast majority assume the existence of diagnostic techniques, and focus on how the availability of information obtained from these diagnoses can be used to develop robust and reliable routing algorithms. This means that diagnoses problem is not addressed by these methods; information about fault location is known in advance; and there are mechanisms to correctly distribute this information to network nodes.

There is some interesting research in the area of component redundancy [4]. On the side of the network reconfiguration approach, there are works based on deterministic routing methodologies for tori and meshes [5], and others that achieve error detection and recovery for k -ary n -cube topologies [6]. In the latter proposal, packet injection is required to be temporarily stopped during a global reconfiguration phase.

In [7] the author proposes a widely used methodology for designing fault-tolerant routing algorithms. Several works have achieved good performance results based on the aforementioned methodology, but almost all of them use a static fault model. When using this model, all the faults need to be known when the system is started. Thus, when a fault occurs, the system has to be stopped

and restarted in order to continue operating. In this line, fault-tolerant routing strategies have been proposed for k -ary n -tree [8] and direct networks [9].

The authors of [9] suggest the use of intermediate nodes to circumvent faults in the interconnection network. Detection of faults, checkpointing, and distribution of routing/fault information are assumed to be provided by the static fault model. The use of such nodes was first proposed by Valiant for the purpose of traffic balancing [10].

A methodology capable to deal with dynamic fault models was proposed in [11] as an evolution of [9]. In order to use the dynamic fault model, status-information must be distributed through control messages and rerouting decisions must then be taken locally based on this information. All these actions have an extra cost that must be considered.

In [12] the authors introduce a fault-tolerant routing methodology that sacrifices a certain number of healthy nodes in order to use no more than two virtual channels, and to keep the time to reach the destination relatively low.

Our work is largely based on the *multipath* concept presented in [13]. Other studies using similar concepts have been published in the last few years and would therefore be interesting to analyze them. In [14] the authors propose the use of several disjoint paths by means of routing tables. A more sophisticated path selection method, where the paths are chosen according to what the authors call “knowledge database of route patterns” was introduced in [15]. A most recent work [16] presents a method for dynamic network reconfiguration. However, this method implements a regressive deadlock recovery approach and therefore package drops are needed. The authors argue that deadlocks are “rare and unlikely to occur” but this argument is not necessarily true, especially when dealing with faults. An enhancement to [13] is presented for congestion control in [17].

The failure of a single network component disables resources for variable time periods, generating congestion problems in their surroundings. Adaptive routing is a natural solution to this problem and therefore the adaptive routing algorithms designed for fault tolerance could be outlined as a viable option. From our point of view, the best approach for tolerating faults in HSIN is by using a fault-tolerant routing method in addition to the four phases of fault tolerance: error detection, damage confinement, error recovery, and fault treatment and service continuity [18].

For this reason, we focus our work on the problem of fault tolerance for HSIN by means of adaptive routing. In this paper we present a method that exploits communication path redundancy through an adaptive multipath routing policy with the aim of solve a certain number of link and node failures. The method is based on source-destination communication path information and consists of three phases. The first phase is responsible for on-line fault diagnosis and uses physical level monitoring at the intermediate nodes along the source-destination path. If a message encounters a faulty link as it progresses towards its destination, the second phase immediately reroutes the message to the destination by an alternative path. In the last phase, the source node is notified about the link

failure in order to disable the faulty path, and to establish new paths for the following messages to be sent to that destination.

Our work supports a dynamic fault model, therefore, the system remains operational while measures are taken to circumvent the faulty components. At the same time, and unlike the earlier mentioned works, deals with the four phases of fault tolerance. However, our method only notifies the event of a failure to the sources nodes that try to send messages by faulty links, instead of distributing status-information over the network. The intermediate nodes selection is based on real-time network traffic conditions in order to maximize performance [17].

Experimental results show an average performance higher than 97% for a set of test scenarios with several faults in a 1024 nodes bidimensional torus network for standard traffic patterns.

The rest of the paper is organized as follows. Section 2 describes the Multipath Fault-Tolerant Routing method and some implementation issues of the work. Evaluation environment, test scenarios and results are presented in Section 3. Finally, in Section 4 some conclusions and future work are drawn.

2 Multipath Fault-Tolerant Routing

The main goal of the Multipath Fault-Tolerant Routing is to prevent network failures in spite of failures in its composing elements: links and nodes. The method differs from the proposals found in the literature through its combined features. In particular, it is able to combine support for a dynamic fault model and fully adaptive routing, while at the same time treating the four phases of fault tolerance. Even more, the method introduces a novelty approach, addressing system continuity functioning and network performance degradation problems at the same time. To accomplish these tasks, it uses a non-global scheme to distribute real-time paths information in order to chose the bests source-destination paths. Furthermore, the method does not require global re-configuration or stopping packet injection at any time, using a limited number of virtual channels.

Conceptually, the method consist of three steps. In the first step the failure in the original path is discovered when a message tries to use a faulty link. In the second step the message is rerouted to its destination through an alternative path. Finally, the source node is notified about the discovery of a link failure in the path, in order to disable the faulty path and reconfigure new path for the following messages. These three steps are illustrated in the Figs. 1(a), 1(b), and 1(c), where the red cross represents a link failure and nodes are represented as S (source) and D (destination).

The method is based on the configuration and use of simultaneous alternative paths between source and destination nodes in order to tolerate link and node failures. The alternative paths are created using intermediates nodes acting like scattering and gathering areas from source and destination nodes. These paths would use the available links in routers and, in some cases, could be non-minimal.

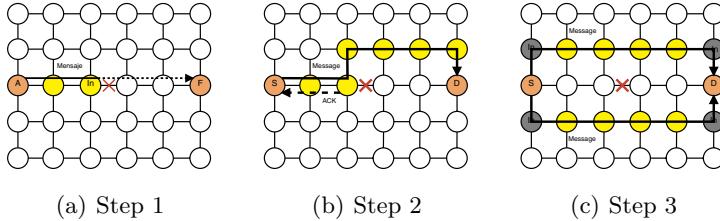


Fig. 1. Example of the method behavior with one link failure

The set of possible alternative paths between each source-destination pair is called *multipath* or *metapath* [13]. An example of a three-line *multipath* is shown in Fig. 2.

The alternative paths are built based on intermediate nodes carefully chosen to ensure that they are not in the original path. These intermediate nodes are surrounding neighbors of source and destination nodes and allow the method to circumvent the faulty areas by means of path segmentation. The intermediate nodes (gray circles in Fig. 2), are chosen according to their distance to the source/destination node. The nodes of 1-hop distance are considered first, then nodes of 2-hop distance, etc. If necessary, e.g. if a link fails, the *multipath* could be expanded in order to include additional alternative paths. This case is shown in Fig. 1, where two alternative paths were included.

In this work, we consider only two intermediate nodes so that the path is divided in three segments: the first ranges from the source (*S*) to the first intermediate node (*In1*), the second between the two intermediate nodes, and the third from the second intermediate node (*In2*) to the destination (*D*). This segmented path is called a *multistep path*, and uses minimal static routing in each segment. When using *multistep paths* deadlock freedom becomes a key issue. In our method, deadlock freedom is ensured by having a separate virtual channel for each step. As we are considering two intermediate nodes, one extra virtual channel is used (if required) from *S* to *In1*, another from *In1* to *In2*, and a third one from *In2* to *D*. This way, each step defines a virtual network, and the packets change virtual network at each intermediate node. Although each virtual network relies on a different virtual channel, they all share the same adaptive channel(s). Therefore, a total of 4 virtual channels are needed. Worthwhile clarify that two virtual channels are already required to provide deadlock-free fully adaptive routing [19].

In Fig. 3 could be seen the behavior of the method, including all its functionalities and operations. The colored blocks in Fig. 3 represents the set of operations performed by the method in the absence of failures while the colorless correspond to the additional features for fault tolerance and congestion control.

When a source injects a message in the interconnection network, it traverses a set of routers before reaching the destination node. Two monitoring actions are conducted at each router along the source-destination path. Link state monitoring is performed directly over router physical channels, while traffic load

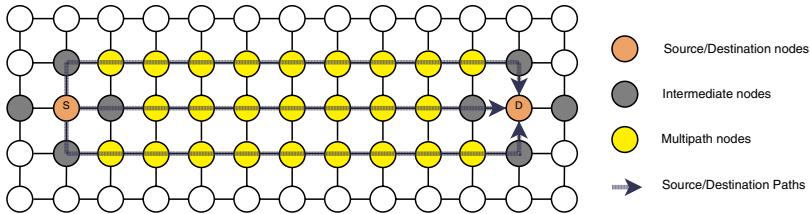


Fig. 2. Example of a three-line multipath

monitoring is accomplished by the router over the message. These monitoring actions are represented in Fig. 3 by the two colored diamonds.

Each message registers and transports the accumulated latency information about the path it traverses. If this latency value exceeds a threshold, the router sends back an acknowledgment message (ACK) in order to alert the source node about a possible congestion situation. Conversely, if the latency value does not exceed the threshold before reaching the destination, an ACK carrying the accumulated latency value is sent back to the source from the destination. The threshold value depends on the program being executed, and it represents the latency that the program can tolerate without its execution being affected.

When a message tries to use a faulty link, two actions are triggered. As a first step, the message is rerouted to its destination through an escape path (*Escape Path Selection* block in Fig. 3). At the same time, a special acknowledgment message is sent back to the source node. This ACK message carries an infinite latency value with the aim of avoid the use of the faulty path by the source node (explained in more detail later). These two triggered actions are graphically shown in Fig. 1(b).

All of these special ACK messages have higher priority in the routing unit, and their size is less than 1% of the data messages because they only transport the latency value.

Just after the ACK message reaches the source node, latency value is used in the dynamic *multipath* configuration phase. This phase, represented by the *Multipath Configuration* block, is responsible for determining the number of alternative paths needed for a specific source-destination pair from the latency values described above. If the latency is below the threshold minus a hysteresis value, the number of paths must be reduced. On the other hand, if the latency is above the threshold plus the hysteresis value, the number of alternative paths must be increased. Note that the number of alternative paths will be increased in the event of a failure due to the infinite latency value.

From this information the *multistep path* selection phase chooses a *multistep path* for each message, avoiding the use of faulty paths, and fairly distributing the communication load over the *multipath*. Messages are distributed in base of each *multistep path* bandwidth (defined as the inverse of the latency). The path with higher bandwidth is most frequently used, then messages are distributed over the *multistep paths* according to their bandwidth proportion. A faulty path is never used because it has been assigned an infinite latency value.

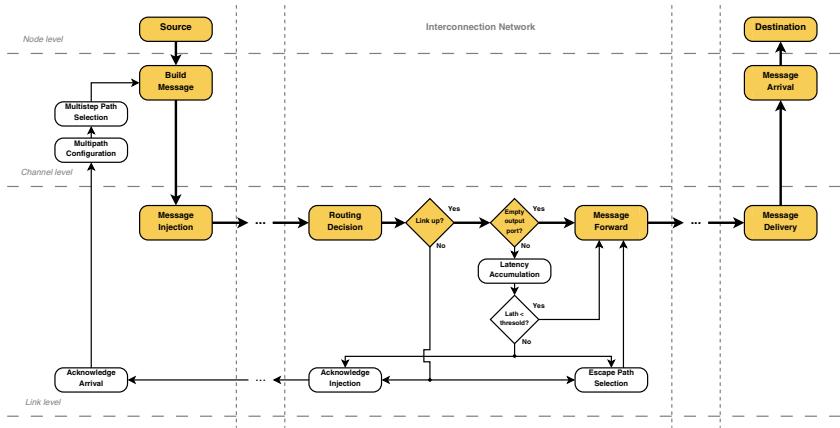


Fig. 3. Multipath Fault-Tolerant Routing behavior diagram

The outcome of this phase is then used by the source node to distribute the load among all the *multistep paths*, avoiding at the same time the faulty paths.

Having described the methodology, some important issues may be addressed. In first place, the scope of the method will be defined. Finally, a brief analysis of costs and resources will be drawn.

Situations where faults completely disconnect network nodes are not considered by our method, and faults are assumed to be fail-stop. These are common assumptions in the context of fault-tolerant routing.

Our method relies on physical level information about links state. This information is already available on almost all modern network devices. Current devices test and control their ports and links by means of physical parameters such as potential difference, impedance, etc. The InfiniBand architecture offers four link states: LinkDown, LinkInitialize, LinkArm and LinkActive [20]. Even the simplest Ethernet router makes available the link state information.

The operations of the method at node level have not a high overhead because these operations are performed locally, they are simple (comparisons and accumulations), and they do not delay send/receive primitives. As shown in Fig. 3, message is forwarded without any overhead when output link is non-faulty. The ACK generation and escape path mechanisms are invoked only when faults or congestion are detected, and their operations are performed when messages are waiting in the queue. Hence, computing these operations is performed concurrently with packet delivery. Furthermore, the interconnection network is usually not planned to continuously operate at its saturation point, thus small overheads could be tolerated if necessary to avoid faults.

3 Performance Evaluation

This section describes the test scenarios used to evaluate our method and, at the same time, provides an explanation of experimental results. Likewise, simulation

environment is briefly explained. The aim of the experiments to be presented in this section is to evaluate the performance behavior of the Multipath Fault-Tolerant Routing method by means of several test scenarios.

The simulation environment is provided by the commercial modeling and simulation tool OPNET Modeler [21]. This tool gives support for modeling communication networks, and allows faults injection in model components. The simulation environment provides a Discrete Event Simulator (DES) engine and offers a hierarchical modeling environment with an enhanced C++ language. Network devices behavior is defined through a Finite State Machine approach (FSM), which supports detailed specification of protocols, resources, applications, algorithms, and queuing policies.

Experimentation is based on direct networks, specifically on two-dimensional torus chosen mainly due to its current popularity and multiple (mostly minimal) alternative paths between nodes. The network was modeled based on interconnection elements, connected among them through links; and endnodes that provide the interface to connect processing nodes to the network. The model was built from previous models for adaptive routing protocols in HSIN [22].

The simulations were conducted for a 1024 nodes network arranged in a 32x32 torus topology. We have assumed Virtual Cut-Through flow control and several standard package sizes with a constant packet injection rate. Link bandwidth was set to 1 Gbps, and the size of routers buffers is 2 MB.

The performance evaluation of the method was measured by means of a set of test scenarios with up to 60 simultaneous random link failures. The experiments have been executed using the standard traffic patterns *Bit reversal*, *Perfect shuffle*, *Butterfly*, *Matrix transpose* and *Complement* [23]. The experiments were simulated several times for each test scenario and its values averaged.

As a first step, a set of fault-free scenarios were simulated in order to get average latency values in the absence of failures. From these values, performance degradation was measured as the difference between average latency values of faulty and fault-free scenarios.

The performance results for the standard traffic patterns in the 60-failures test scenario are shown in Fig. 4. To ease the interpretation of results, the *y* axis is only visualized in the range [80:100], and pattern traffics were label as follows: Bit reversal (BR), Perfect shuffle (PS), Butterfly (BF), Matrix transpose (MT) and Complement (CM).

In order to compare the ratio between the number of failures and the method performance, the results of the 6-failures test scenario are also included and shown in Fig. 5.

As shown in Figs. 4 and 5, our method obtains very high performance levels. An important point to emphasize is the fact that with a linear increase of 10 times the number of faults, the average performance degradation value is just about 3%. In the worst case the performance is about 88% for the *Perfect shuffle* pattern with 60 simultaneous faults, and 100% in the best case for *Matrix transpose* pattern with 6 simultaneous faults. Performance values are even better if

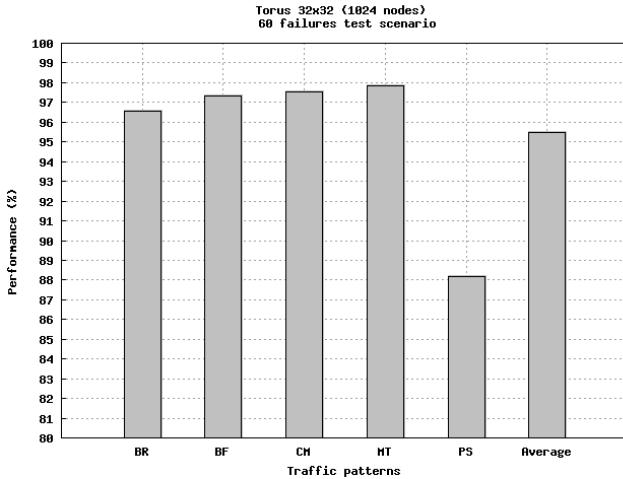


Fig. 4. Sixty-failures test scenario results

we consider the average values, obtaining a 97% performance value in the worst case (60 simultaneous faults).

From these results we can conclude that our fault-tolerant routing policy achieves very good performance values with minimal degradations for all the test scenarios evaluated in this work.

4 Conclusions

In this paper, we have proposed a fault-tolerant routing method designed to deal with the fault tolerance problem for High-Speed Interconnection Networks. This method is based on an adaptive routing approach and, to the best of our knowledge, is the only one that treats the four phases of fault tolerance from this approach. In particular, it is able to support a dynamic fault model, while at the same time not requiring network reconfigurations or stopping packet injection at any time, using a limited number of virtual channels. The method needs few additional hardware resources and is able to treat the intermittent and transient faults as well as the permanent ones, maximizing the resources utilization. Unlike other fault-tolerant approaches, our method does not degrade at all the system performance in the absence of faults. Moreover, one of the most important features of the methodology is the fact that it doesn't lose any message in the presence of faults (given the scenarios tolerated by the method).

Evaluation results show an average performance value higher than 97% for several test scenarios ranging from 1 up to 60 number of faults, using the standard communication patterns. From these results we conclude that our method is capable of reroute messages to their destinations through fault-free paths with a negligible performance degradation even in the presence of a high number of faults.

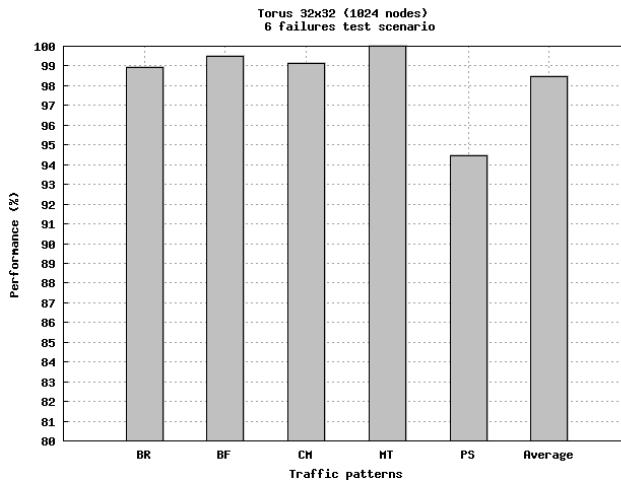


Fig. 5. Six-failures test scenario results

Future work includes the expansion of the current fault models in order to address information losses and stuck caused by the failures of network devices. Also, we plan to study influence of failures when network is operated near saturation point in order to design fault tolerant methods suitable to this situation.

References

1. Barroso, L., Dean, J., Holzle, U.: Web search for a planet: The google cluster architecture. *IEEE Micro.* 23(2), 22–28 (2003)
2. Adiga, N., Almasi, G., Almasi, G., Aridor, Y., Barik, R., et al.: An overview of the BlueGene/L supercomputer. In: *Supercomputing ACM/IEEE 2002 Conference*, November 2002, p. 60 (2002)
3. Abd-El-Barr, M.: Design and analysis of reliable and fault-tolerant computer systems. Imperial College Press, London (2007)
4. Sem-Jacobsen, F., Skeie, T., Lysne, O., et al.: Siamese-twin: A dynamically fault-tolerant fat-tree. In: *International Parallel and Distributed Processing Symposium (IPDPS 2005)*, April 2005, p. 100b. IEEE Computer Society Press, Los Alamitos (2005)
5. Mejia, A., Flich, J., Duato, J., Reinemo, S.A., Skeie, T.: Segment-based routing: an efficient fault-tolerant routing algorithm for meshes and tori. In: *International Parallel and Distributed Processing Symposium (IPDPS 2006)*, April 2006, p. 10. IEEE Computer Society Press, Los Alamitos (2006)
6. Puente, V., Gregorio, J.A.: Immucube: Scalable fault-tolerant routing for k-ary n-cube networks. *IEEE Transactions on Parallel and Distributed Systems* 18(6), 776–788 (2007)
7. Duato, J.: A theory of fault-tolerant routing in wormhole networks. *IEEE Transactions on Parallel and Distributed Systems* 8(8), 790–802 (1997)

8. Gómez, C., Gómez, M.E., López, P., Duato, J.: An efficient fault-tolerant routing methodology for fat-tree interconnection networks. In: Stojmenovic, I., Thulasiram, R.K., Yang, L.T., Jia, W., Guo, M., de Mello, R.F. (eds.) ISPA 2007. LNCS, vol. 4742, pp. 509–522. Springer, Heidelberg (2007)
9. Gómez, M.E., Nordbotten, N.A., Flich, J., López, P., Robles, A., Duato, J., Skeie, T., Lysne, O.: A routing methodology for achieving fault tolerance in direct networks. *IEEE Transactions on Computers* 55(4), 400–415 (2006)
10. Valiant, L.G., Brebner, G.J.: Universal schemes for parallel communication. In: STOC 1981: Proceedings of the thirteenth annual ACM symposium on Theory of computing, pp. 263–277. ACM, New York (1981)
11. Nordbotten, N.A., Skeie, T.: A routing methodology for dynamic fault tolerance in meshes and tori. In: Aluru, S., Parashar, M., Badrinath, R., Prasanna, V.K. (eds.) HiPC 2007. LNCS, vol. 4873, pp. 514–527. Springer, Heidelberg (2007)
12. Ho, C.-T., Stockmeyer, L.: A new approach to fault-tolerant wormhole routing for mesh-connected parallel computers. *IEEE Transactions on Computers* 53(4), 427–438 (2004)
13. Franco, D., Garcés, I., Luque, E.: Distributed routing balancing for interconnection network communication. In: HIPC 1998. 5th International Conference On High Performance Computing, pp. 253–261 (1998)
14. Montañaña, J.M., Flich, J., Robles, A., Lopez, P., Duato, J.: A transition-based fault-tolerant routing methodology for infiniband networks. In: Communication Architecture for Clusters 2004 (CAC 2004), International Parallel and Distributed Processing Symposium (IPDPS 2004), Santa Fe, New Mexico, USA, April 2004, p. 186. IEEE Computer Society Press, Los Alamitos (2004)
15. Montañaña, J.M., Flich, J., Robles, A., Duato, J.: A scalable methodology for computing fault-free paths in infiniBand torus networks. In: Labarta, J., Joe, K., Sato, T. (eds.) ISHPC 2006 and ALPS 2006. LNCS, vol. 4759, pp. 79–92. Springer, Heidelberg (2008)
16. Montañaña, J., Flich, J., Duato, J.: Epoch-based reconfiguration: Fast, simple, and effective dynamic network reconfiguration. In: International Parallel and Distributed Processing Symposium (IPDPS 2008), April 2008, pp. 1–12. IEEE Computer Society, Los Alamitos (2008)
17. Lugones, D., Franco, D., Luque, E.: Dynamic and distributed multipath routing policy for High-Speed cluster networks. In: 9th IEEE/ACM International Symposium on Cluster Computing and the Grid - CCGRID 2009, Shanghai, China (May 2009)
18. Jalote, P.: Fault tolerance in distributed systems. PTR Prentice Hall, Englewood Cliffs (1994)
19. Dao, B.V., Duato, J., Yalamanchili, S.: Dynamically configurable message flow control for fault-tolerant routing. *IEEE Transactions on Parallel and Distributed Systems* 10(1), 7–22 (1999)
20. InfiniBand Trade Association: InfiniBand architecture specification: release 1.2., vol. 1. InfiniBand Trade Association, Portland, OR (2004)
21. OPNET Technologies: Opnet modeler. (January 2009), <http://www.opnet.com/>
22. Lugones, D., Franco, D., Luque, E.: Modeling adaptive routing protocols in high speed interconnection networks. In: OPNETWORK 2008 Conference (2008)
23. Duato, J., Yalamanchili, S., Ni, L.M.: Interconnection networks. In: An Engineering Approach, Morgan Kaufmann, San Francisco (2003)

Hardware Implementation Study of the SCFQ-CA and DRR-CA Scheduling Algorithms*

Raúl Martínez¹, Francisco J. Alfaro², José L. Sánchez², and José M. Claver³

¹ Intel-UPC Barcelona Research Center,
Barcelona, SPAIN 08034
raulmm@dsi.uclm.es

² Dpto. de Sistemas Informáticos. Univ. Castilla-La Mancha
Albacete, SPAIN 02071

{falfaro,jsanchez}@dsi.uclm.es

³ Dpto de Informática. Univ. Valencia
Valencia, SPAIN 46100

jclaver@uv.es

Abstract. The provision of Quality of Service (QoS) in computing and communication environments has been the focus of much research in industry and academia during the last decades. A key component for networks with QoS support is the egress link scheduling algorithm. Apart from providing a good performance in terms of, for example, good end-to-end delay and fair bandwidth allocation, an ideal scheduling algorithm implemented in a high-performance network with QoS support should satisfy other important property which is to have a low computational and implementation complexity. This is especially important in high-performance networks due to their high speed and because switches are usually implemented in a single chip.

In [7] we proposed the Self-Clocked Fair Queuing Credit Aware (SCFQ-CA) and the Deficit Round Robin Credit Aware (DRR-CA) schedulers in order to adapt the SCFQ and DRR algorithms to networks with a link-level flow control mechanism. In this paper, we propose specific implementations of these two schedulers taking into account the characteristics of current high-performance networks. Moreover, we compare the complexity of these two algorithms in terms of silicon area and computation delay. In order to carry out this comparison, we have performed our own hardware implementation for the different schedulers. We have modeled the schedulers using the Handel-C language and employed the DK design suite tool from Celoxica in order to obtain hardware estimates on silicon area and arbitration time.

* This work has been jointly supported by the Spanish MEC and European Comission FEDER funds under grants Consolider Ingenio-2010 CSD2006-00046 and TIN2006-15516-C04-02 and by Junta de Comunidades de Castilla-La Mancha under grant PCC08-0078-9856. Raúl Martínez was with the University of Castilla-La Mancha when the main ideas of the paper where developed.

1 Introduction

The advent of high-speed networking has introduced opportunities for new applications. Current packet networks are required to carry not only traffic of applications, such as e-mail or file transfer, which does not require pre-specified service guarantees, but also traffic of other applications that requires different performance guarantees, like real-time video or telecommunications [8]. Even in the same application, different kinds of traffic (e.g. I/O requests, coherence control messages, synchronization and communication messages, etc.) can be considered, and it would be very interesting that they were treated according to their priority [3].

The provision of QoS in computing and communication environments has been the focus of much discussion and research in academia during the last decades. This interest in academia has been renewed by the growing interest on this topic in industry during the last years. A sign of this growing interest in industry is the inclusion of mechanisms intended for providing QoS in some of the last network standards like Gigabit Ethernet, InfiniBand, or Advanced Switching (AS). An interesting survey with the QoS capabilities of these network technologies can be found in [10]. Common characteristics of the specifications of these network technologies intended to provide QoS are the use of a link-level flow control mechanism, which makes the networks lossless, a reduced set of Virtual Channels (VCs), and an egress link scheduler to arbitrate among the traffic transmitted in each VC. These last two mechanisms permit us to aggregate traffic with similar characteristics in the same VC and to provide each VC with a different treatment according to its requirements, at the style of the differentiated services (DiffServ) QoS model [1].

A key component for networks with QoS support is the output (or egress link) scheduling algorithm (also called service discipline), which selects the next packet to be transmitted and decides when it should be transmitted [4], [18]. A lot of possible scheduling algorithms have been proposed in the literature. However, most of these algorithms were proposed for the Internet scenario with a high number of possible flows, without taking into account a link-level flow control mechanism, and were intended mainly to be implemented by software. These characteristics differ from the requirements of current high-performance networks. In [7], we already gave a first step in adapting some scheduling algorithms to high-performance networks by proposing new versions of some schedulers in order to support a link-level flow control mechanism.

Apart from providing a good performance in terms of, for example, good end-to-end delay (also called latency) and fair bandwidth allocation, an ideal scheduling algorithm implemented in a high-performance network with QoS support should satisfy other important property which is to have a low computational and implementation complexity [13]. We can measure the complexity of a scheduler based on two parameters: Silicon area required to implement the scheduling mechanism and time required to determine the next packet to be transmitted. A short scheduling time is an efficiency requirement that takes more importance in high-performance networks due to their high speed. Moreover, switches of

high-performance interconnection technologies are usually implemented in a single chip. Therefore, the silicon area required to implement the various switch elements is a key design feature. Note that the scenario that we are addressing here is very different than for example IP routers with QoS support, where these algorithms are usually implemented by software instead of hardware.

The fair queuing algorithms are an important kind of scheduling algorithms that allocate bandwidth to the different flows in proportion to a specified set of weights. The perfect fair queuing scheduling algorithm is the General Processor Sharing (GPS) scheduler [4], [9], which is based on a fluid model that provides perfect instant fairness in bandwidth allocation and has many desirable properties [15], [9]. Different packet-by-packet approximations of GPS have been proposed, which try to emulate the GPS system as accurately and simply as possible while still treating packets as entities.

The “Sorted-priority” family of algorithms, which includes the Weighted Fair Queuing (WFQ) [4] and Self-Clock Fair Queueing SCFQ [6] algorithms, are known to offer good delay bounds [14]. This kind of scheduling algorithms assign each packet a tag and the scheduling is made based on the ordering of these tags. The main complexity factors in this kind of schedulers comes from tag calculation and tag sorting. Note that these algorithms were proposed mainly for the Internet scenario with a high number of possible flows. Moreover, a common problem in the sorted-priority approach is that tags cannot be reinitialized to zero until the system is completely empty and all the sessions are idle. The reason is that these tags depend on a common-reference virtual clock and are an increasing function of the time. In other words, it is impossible to reinitialize the virtual clock during the busy period, which, although statistically finite (if the traffic is constrained), can be extremely long, especially given that most communication traffic exhibits self-similar patterns which lead to heavily tailed buffer occupancy distributions.

To avoid the complexity of the sorted-priority approach, the Deficit Round Robin (DRR) algorithm [12] has been proposed. The aim of DRR is to implement fair queuing and achieve practically acceptable complexity at the expense of other performance metrics such as fairness and delay. Due to its computational simplicity, recent research in the Differentiated Services area proposes the DRR as a feasible solution for implementing the Expedited Forwarding Per-hop Behavior [5].

In this paper, we propose specific implementations (taking into account the characteristics of current high performance networks) of the SCFQ and DRR scheduling algorithms and compare their complexity in terms of silicon area and computation delay. In fact, we are going to consider the modified versions of these two algorithms that we proposed in [7] in order to maintain the fair bandwidth allocation in networks with a link-level flow control network, which is the case in most current high-performance network technologies. We have chosen the SCFQ algorithm as an example of “sorted-priority” algorithm and the DRR algorithm because of its very small computational complexity.

In [16] and [11] interesting implementations for the SCFQ scheduler are proposed. However, these implementations were designed for a high number of possible flows. Note that in our case there is going to be just a limited number

of VCs. This allows us to consider more efficient implementations. Moreover, the case of the SCFQ implementation [11] was intended for fixed packet sizes, specifically, for an ATM environment.

Therefore, we have performed our own hardware implementation for the different schedulers. We have modeled the schedulers using the Handel-C language [2] and employed the DK design suite tool from Celoxica in order to obtain hardware estimates on silicon area and arbitration time.

The structure of the paper is as follows: Sections 2 and 3 present the DRR-CA and SCFQ-CA scheduling algorithms, respectively, state how we have adapted them to current high-performance network characteristics, and describe specific hardware implementation decisions. In Section 4 a comparison study on the implementation and computational complexity of the two schedulers is provided. Finally, some conclusions are given.

2 Implementation of the SCFQ-CA Scheduler

The Self-Clocked Weighted Fair Queuing (SCFQ) algorithm [6] is a variant of the well known Weighted Fair Queuing (WFQ) mechanism [4] which has a lower computational complexity. This objective is accomplished by adopting a different notion of the virtual time employed in WFQ. Instead of linking the virtual time to the work progress in the GPS system, the SCFQ algorithm uses a virtual time function which depends on the progress of the work in the actual packet-based queuing system. This approach offers the advantage of removing the computation complexity associated to the evaluation of the virtual time that may make WFQ unfeasible in high-speed interconnection technologies.

Therefore, when a packet arrives, SCFQ uses the service tag of the packet currently in service as the virtual time to calculate the new packet tag. Thus, in this case the service tag of the k^{th} packet of the i^{th} flow is computed as

$$S_i^k = \max\{S_i^{k-1}, S_{current}\} + \frac{L_i^k}{\phi_i},$$

where L_i^k is the packet length, ϕ_i the weight assigned to its flow, and $S_{current}$ is the service tag of the packet being serviced. Figure 1 shows the pseudocode for the SCFQ algorithm.

In [7] we proposed a new version of this scheduler, the SCFQ Credit Aware (SCFQ-CA) algorithm, that takes into account the link level flow control mechanism and VCs instead of flows. The SCFQ-CA algorithm that we proposed works in the same way as the SCFQ algorithm, except in the following aspects:

- When a new packet arrives at a VC queue, a service tag is assigned only if the arrived packet is at the head of the VC and there are enough credits to transmit it.
- When a packet is transmitted, if there are enough credits to transmit the next packet, the VC service tag is recalculated.
- When a VC is inactive due to a lack of credits and receives enough credits to transmit again, a new service tag is assigned to the VC.

```

PACKET ARRIVAL (newPacket, flow):

$$\begin{aligned} newPacket_{serviceTag} &\leftarrow \max(currentServiceTag, flow_{lastServiceTag}) \\ &\quad + \frac{newPacket_{size}}{flow_{reservedBandwidth}} \\ flow_{lastServiceTag} &\leftarrow newPacket_{serviceTag} \end{aligned}$$


ARBITRATION:
while (There is at least one packet to transmit)
  selectedPacket  $\leftarrow$  Packet with the minimum serviceTag
  currentServiceTag  $\leftarrow$  selectedPacketserviceTag
  Transmit selectedPacket
  if (There are no more packets to transmit)
     $\forall flow$  flowlastServiceTag  $\leftarrow$  0
    currentServiceTag  $\leftarrow$  0

```

Fig. 1. Pseudocode of the SCFQ scheduler

Note that once that there is at least one packet in a VC queue, the value of the service tags of the packets that arrive after this first packet depends only on the value of the precedent service tags and not on the value of $S_{current}$ at the arrival time. Therefore, we can wait to stamp a packet until it arrives at the head of a VC. This allows us to simplify in a high degree the original SCFQ algorithm by storing not a service tag per packet, but a service tag per flow or VC. This service tag represents the service tag of the packet at the head of the VC queue. Note that this makes much easier and simpler to modify this algorithm to take into account a link-level flow control mechanism. Each VC service tag is then computed as:

$$S_i = S_{current} + \frac{L_i^{first}}{\phi_i},$$

where L_i^{first} is the size of the packet at the head of the i^{th} VC.

With these modifications, we have already taken into account the presence of an egress link scheduler. However, we can take more advantage of the limited number of VCs to simplify the actual implementation. The SCFQ-CA algorithm, as the original SCFQ algorithm and most sorted-priority algorithms, still has the problem of the increasing tag values and the possible overflow of the registers used to store these values. Therefore, we propose a modification to the SCFQ-CA scheduler that makes impossible this overflow. This modification consists in subtracting the service tag of the packet currently being transmitted to the rest of service tags. If we consider only a tag per VC, this means to subtract the service tag of the VC to which the packet being transmitted belongs to the rest of VCs service tags. This limits the maximum value of the service tags while still maintaining the absolute differences among their values. This also means that $S_{current}$ is always equal to zero and thus,

$$S_i = \frac{L_i^{first}}{\phi_i}.$$

Moreover, the service tags are limited to a maximum value max_S : $max_S = \frac{MTU}{min_\phi}$ where MTU is the maximum packet size and min_ϕ is the minimum possible weight that can be assigned to a VC. The resulting SCFQ-CA scheduling algorithm is represented in the pseudocode shown in Figure 2. Note that this last modification adds the complexity of subtracting to all the service tags a certain value each time a packet is scheduled. This makes this modification feasible in hardware only when a few number of VCs is considered, which is the common trend in the last high-performance network proposals.

```

PACKET ARRIVAL(newPacket, VC):
if (newPacket is at the head in the queue of VC) and
    (The flow control does allow transmitting from VC))
     $VC_{serviceTag} \leftarrow \frac{VC_{sizeFirst}}{VC_{reservedBandwidth}}$ 
ARBITRATION:
while (There is at least one active VC)
    selectedVC  $\leftarrow$  Active VC with the minimum serviceTag
    currentServiceTag  $\leftarrow$  selectedVCserviceTag
    Transmit packet from selectedVC
     $\forall$  active VC
         $VC_{serviceTag} \leftarrow VC_{serviceTag} - currentServiceTag$ 
    if ((There are more packets in the queue of selectedVC) and
        (The flow control does allow transmitting from selectedVC))
        selectedVCserviceTag  $\leftarrow \frac{selectedVC_{sizeFirst}}{selectedVC_{reservedBandwidth}}$ 

```

Fig. 2. Pseudocode of the improved SCFQ-CA scheduler

In order to implement this scheduler, when a new packet arrives at the SCFQ-CA scheduler, apart from taking note of the packet size and activating the VC if there are enough flow control credits to transmit that packet, this scheduler must calculate the packet service tag. As stated before, we have solved the problem of the possible overflow of the service tags. Moreover, this modification entails a simplification of the service tag computation. In order to decide which is the next packet to be transmitted, the SCFQ-CA algorithm must choose the packet from the active VC with the smallest service tag. In order to do this in an efficient way, we have employed a bitonic network, which obtains the selected VC in $\log(\#VCs)$ cycles. The structure of the selector module is shown in Figure 3.

Note that, in order to calculate the packet service tag a division operation is performed. This operation requires an arithmetical unit that is not simple at all. Handel-C, which is the language that we have used to model the schedulers, offers a divisor operand that calculates the result in one cycle (as all the Handel-C

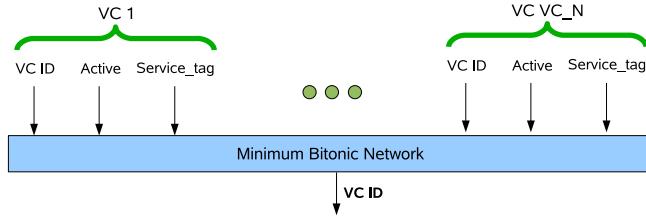


Fig. 3. Structure of the selector module for the SCFQ-CA scheduler

statements). This operand makes the division very short in terms of number of cycles but, it makes the cycle time very long, and thus it makes the arbitration time quite long. Therefore, we have employed a mathematical division unit that performs the division in several cycles. Specifically, it takes a number of cycles equal to the length of the operators plus one. With this new version the calculation of the time tag requires more cycles to be performed but reduces the cycle time and thus, the arbitration time, which is a more critical value. Therefore, the estimates that we show in Section 4 have been obtained using this second version.

3 Implementation of the DRR-CA Scheduler

The Deficit Round Robin (DRR) algorithm [2] is a variation of the Weighted Round Robin (WRR) algorithm that works on a proper way with variable packet sizes. In order to handle properly variable packet sizes, the DRR algorithm associates each queue with a *quantum* and a *deficit counter*. The quantum assigned to a flow is proportional to the bandwidth assigned to that flow. The sum of all

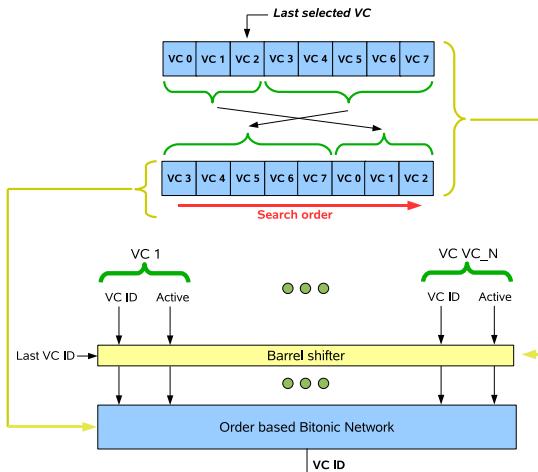


Fig. 4. Structure of the next VC to transmit selector in the DRR-CA scheduler

the quantums is called the frame length. For each flow, the scheduler transmits as many packets as the quantum allows. When a packet is transmitted, the quantum is reduced by the packet size. The unused quantum is saved in the deficit counter, representing the amount of quantum that the scheduler owes the flow. At the next round, the scheduler will add the previously saved quantum to the current quantum. When the queue has no packets to transmit, the quantum is discarded, since the flow has wasted its opportunity to transmit packets.

The DRR Credit Aware (DRR-CA) algorithm that we proposed in [7] works in the same way as the DRR algorithm, except in the following aspects:

- A VC queue is considered active only if it has at least one packet to transmit and if there are enough credits to transmit the packet at the head of the VC.
- When a packet is transmitted, the next active VC is selected when any of the following conditions occurs:
 - There are no more packets from the current VC or there are not enough flow control credits for transmitting the packet that is at the head of the VC. In any of these two cases, the current VC becomes inactive, and its deficit counter becomes zero.
 - The remaining quantum is less than the size of the packet at the head of the current VC. In this case, its deficit counter becomes equal to the accumulated weight in that instant.

A possible way of implementing the mechanism that selects the next active VC would be to check sequentially all the VCs in the list starting from the contiguous position of the last selected VC. However, in order to make this search efficient, we have implemented it with a *barrel shifter* connected to an *order based bitonic network*. The barrel shifter rearranges the list in the correct order of search and the bitonic network finds the first active VC in a logarithmic number of cycles. The structure for this selector function is shown in Figure 4.

4 Hardware Estimates

In this section we analyze the implementation and computational complexity of the DRR-CA and SCFQ-CA schedulers. We have modeled these schedulers using Handel-C language [2] and employed the DK design suite tool from Celoxica in order to obtain hardware estimates on silicon area and arbitration time. Handel-C's level of design abstraction is above Register Transfer Level (RTL) languages, like VHDL and Verilog, but below behavioral. In Handel-C each assignment takes one clock cycle to complete, so it is not a behavioral language in terms of timing.

The source code completely describes the execution sequence and the most complex expression determines the clock period. Note that the Handel-C code that we have designed can actually be used to implement the schedulers in a Field Programmable Gate Array (FPGA) or, if the appropriate conversion is made, in an Application Specific Integrated Circuit (ASIC). However, this has not been the objective of our work, but to obtain the relative differences on

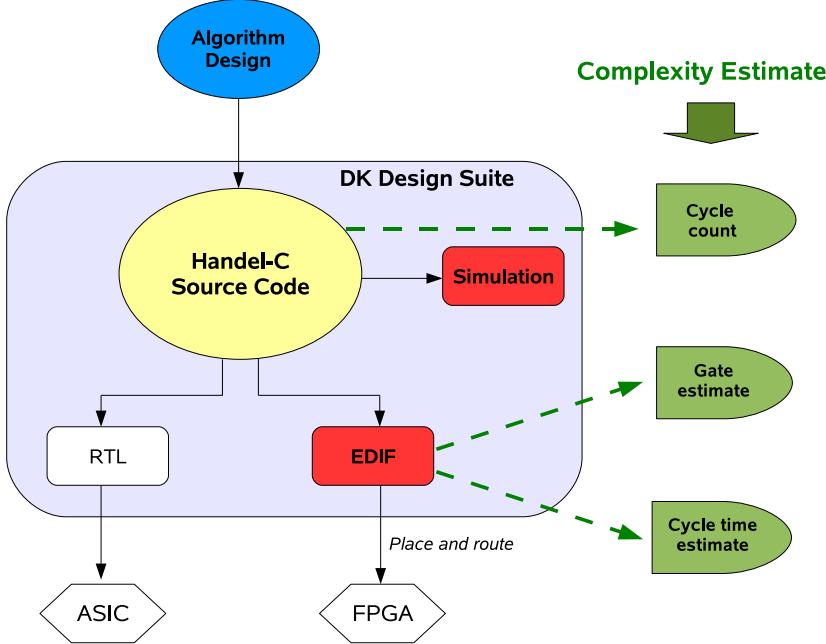


Fig. 5. Design flow with DK employing Handel-C

silicon area and arbitration time for the different schedulers and to measure the effect of the number of VCs and the Maximum Transfer Unit (MTU).

In order to obtain the hardware estimates in which we are interested:

1. We have modeled in Handel-C a full egress queuing system, which includes the scheduling mechanism.
2. We have validated the schedulers employing the simulation and debugging functionality of the DK design suite.
3. We have isolated the scheduler module in order to obtain estimates without influence of other modules.
4. We have obtained the Electronic Design Interchange Format (EDIF) output for a Virtex 4 FPGA from Xilinx [17].

A cycle count is available from the Handel-C source code: Each statement in the Handel-C source code is executed in a single cycle in the resulting hardware design and thus, the number of cycles required to perform a given function can be deduced directly from the source code. Moreover, an estimate of gate count and cycle time is generated by the EDIF Handel-C compiler. The cycle time estimate is totally dependent on the specific target FPGA, in this case the Virtex 4 [17]. However, as our objective is to obtain relative values instead of absolute ones, we consider that this approach is good enough to be able to compare the complexity of the different schedulers. Figure 5 reflects the design flow that we have followed.

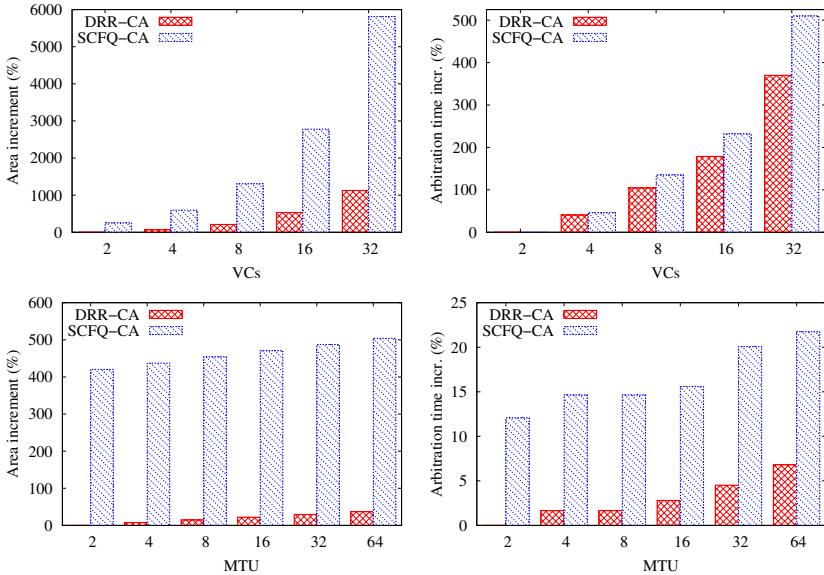


Fig. 6. Comparison of the silicon area and arbitration time required by the DRR-CA and the SCFQ-CA schedulers for different number of VCs and MTU

Figure 6 shows a comparison of the DRR-CA and SCFQ-CA algorithms and how the increment in the number of VCs and the MTU affects the complexity of the two schedulers. Specifically, we have varied the number of VCs with a fixed MTU of 32 and the MTU value with the number of VCs fixed to 8. The figure shows the percentage of increment in silicon area and arbitration time respect to the silicon area and arbitration time required by the DRR-CA scheduler with 2 VCs (when varying the number of VCs) and a MTU of 2 (when varying the MTU).

Regarding the effect of the number of VCs, Figure 6 shows that this number influences dramatically the silicon area and arbitration time required by the DRR-CA and SCFQ-CA schedulers. Note that in the case of the arbitration time, the increment is due to both, the increase in the cycle time and the increase in the number of cycles required to compute the arbitration. Note also that the X axis is in logarithmic scale, thus a linear growth in data plot actually means a logarithmic data growth, and an exponential growth in data plot actually means a linear data growth. On the other hand, regarding the effect of the MTU, Figure 6 shows that the increase in silicon area and time when increasing the MTU is not so important if compared with the effect of the number of VCs.

Finally, this figure shows, as expected, that the DRR-CA scheduler is the simplest scheduler in terms of silicon area and arbitration time. The SCFQ-CA scheduler requires quite more silicon area than the DRR-CA scheduler. However, the difference in arbitration time is not so big. Nevertheless, as was shown in 7 for multimedia traffic the SCFQ-CA scheduler is able to provide much more tight QoS requirements than the DRR-CA scheduler. These results are

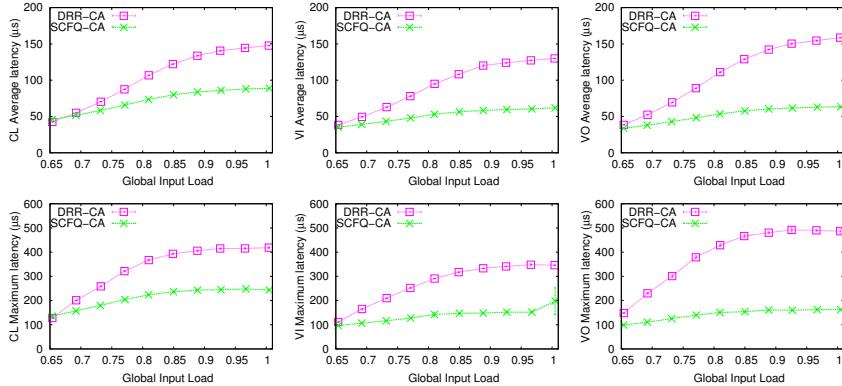


Fig. 7. Latency comparison for controlled load (CL), video (VI), and voice (VO) traffic, for the DRR-CA and SCFQ-CA schedulers

depicted again in Figure 7, which shows the performance provided by the DRR-CA and SCFQ-CA schedulers to three types of multimedia traffic that are simultaneously injected into the network with different amounts of best-effort traffic.

5 Conclusions

Current high-performance networks with QoS support require egress link scheduling algorithms to be fast and simple. Moreover, most well-known algorithms were designed for managing a lot of flows in lossy networks. Therefore, it is necessary to propose specific implementations of the scheduling algorithms adapted to current high-performance requirements. In this work we have proposed implementations and performed a complexity study for two fair queueing scheduling algorithms, the DRR-CA and SCFQ-CA schedulers, taking into account these requirements, and with the objective of fulfilling the complexity constraints in high-performance networks.

The complexity estimates that we have obtained show that the SCFQ-CA scheduler is quite more complex than the DRR-CA in terms of silicon area. However, this algorithm provides much better performance in terms of latency. Therefore, the decision of choosing one of these algorithms for the switches of a real network will be determined by how sensitive to the QoS requirements are the applications that are expected to traverse that network and by the area that there is available to implement the schedulers at the egress links.

References

1. Blake, S., Back, D., Carlson, M., Davies, E., Wang, Z., Weiss, W.: An Architecture for Differentiated Services. Internet Request for Comment RFC 2475, Internet Engineering Task Force (December 1998)
2. Celoxica. Handel-C Language Reference Manual for DK4 (2005)

3. Cheng, L., Muralimanohar, N., Ramani, K., Balasubramonian, R., Carter, J.B.: Interconnect-aware coherence protocols for chip multiprocessors. In: ISCA, pp. 339–351. IEEE Computer Society, Los Alamitos (2006)
4. Demers, A., Keshav, S., Shenker, S.: Analysis and simulations of a fair queuing algorithm. In: SIGCOMM (1989)
5. Charny, A., et al.: Supplemental information for the new definition of EF PHB (Expedited Forwarding Per-Hop-Behavior). RFC 3247 (March 2002)
6. Golestani, S.J.: A self-clocked fair queueing scheme for broadband applications. In: INFOCOM (1994)
7. Martínez, R., Alfaro, F.J., Sánchez, J.L.: A framework to provide quality of service over Advanced Switching. IEEE Transactions on Parallel and Distributed Systems 19(8), 1111–1123 (2008)
8. Montessoro, P.L., Pierattoni, D.: Advanced research issues for tomorrow's multimedia networks. In: International Symposium on Information Technology, ITCC (2001)
9. Parekh, A.K., Gallager, R.G.: A generalized processor sharing approach to flow control in integrated services networks: The single-node case. In: IEEE/ACM Transactions on Networking (1993)
10. Reinemo, S.A., Skeie, T., Sødring, T., Lysne, O., Trudbakken, O.: An overview of QoS capabilities in InfiniBand, Advanced Switching Interconnect, and Ethernet. IEEE Communications Magazine 44(7), 32–38 (2006)
11. Rexford, J., Greenberg, A.G., Bonomi, F.: Hardware-efficient fair queueing architectures for high-speed networks. In: INFOCOM (2), pp. 638–646 (1996)
12. Shreedhar, M., Varghese, G.: Efficient fair queueing using deficit round robin. In: SIGCOMM, pp. 231–242 (1995)
13. Sivaraman, V.: End-to-End delay service in high speed packet networks using Earliest Deadline First Scheduling. PhD thesis, University of California (2000)
14. Stiliadis, D., Varma, A.: Latency-rate servers: A general model for analysis of traffic scheduling algorithms. In: IEEE/ACM Transactions on Networking (1998)
15. Turner, J.S.: New directions in communications (or which way to the information age). IEEE Communications 24(10), 8–15 (1986)
16. Vellore, P., Venkatesan, R.: Performance analysis of scheduling disciplines in hardware. In: Canadian Conference on Electrical and Computer Engineering (CCECE) (May 2004)
17. Xilinx. Virtex-4 family overview. Fact sheet DS112 (v2.0) (June 2007)
18. Zhang, H.: Service disciplines for guaranteed performance service in packet-switching networks (1995)

Topic 14

Mobile and Ubiquitous Computing

Introduction

Gerd Kortuem*, Henk Eertink*, Christian Prehofer*, and Martin Strohbach*

The convergence of wireless networks with mobile devices and embedded sensing technologies has a tremendous impact on how to model, develop and evaluate computer and software systems. Consequently, new computing and system models such as context-aware computing, spatially-aware computing and physically-embedded computing are supplementing or replacing traditional notions of distributed and parallel computing. This topic explores fundamental and practical research questions related to these emerging concepts and mobile and ubiquitous computing in general. This topic is new in the scope of the EuroPar series of conferences. It attracted 4 submissions, of which one was accepted, reflecting a 25% acceptance rate. This particular paper is quite representative for the ubiquitous computing area: it presents solutions for (near) optimal broadcasting, focusing on energy-efficiency in particular. This is very important for battery-powered wireless sensor networks, where broadcasting is the main communication mechanism, and energy efficiency is the key deployment requirement.

* Topic Chairs.

Optimal and Near-Optimal Energy-Efficient Broadcasting in Wireless Networks

Christos A. Papageorgiou¹, Panagiotis C. Kokkinos^{1,2},
and Emmanouel A. Varvarigos^{1,2}

¹ Department of Computer Engineering and Informatics, University of Patras, Greece

² Research Academic Computer Technology Institute, P.O. Box 1122, 26110 Patras, Greece

Abstract. In this paper we propose an energy-efficient broadcast algorithm for wireless networks for the case where the transmission powers of the nodes are fixed. Our algorithm is based on the multicost approach and selects an optimal energy-efficient set of nodes for broadcasting, taking into account: i) the node residual energies, ii) the transmission powers used by the nodes, and iii) the set of nodes that are covered by a specific schedule. Our algorithm is optimal, in the sense that it can optimize any desired function of the total power consumed by the broadcasting task and the minimum of the current residual energies of the nodes, provided that the optimization function is monotonic in each of these parameters. Our algorithm has non-polynomial complexity, thus, we propose a relaxation producing a near-optimal solution in polynomial time. Using simulations we show that the proposed algorithms outperform other established solutions for energy-aware broadcasting with respect to both energy consumption and network lifetime. Moreover, it is shown that the near-optimal multicost algorithm obtains most of the performance benefits of the optimal multicost algorithm at a smaller computational overhead.

1 Introduction

The cooperative nature of both ad hoc and sensor networks, makes broadcasting one of the most frequently performed primitive communication tasks, used for example in order to disseminate topology information and data collected by the sensors, respectively for each class of networks. Being able to perform these communication tasks in an energy-efficient manner is an important priority for such networks.

We study broadcasting in static wireless networks consisting of nodes that have different but fixed levels of transmission power. Much of the previous related work assumes that nodes are able to adjust their transmission power to any desired level. It is quite common that the nodes comprising ad hoc or sensor networks are not able to dynamically adjust their transmission power, since their processing capabilities are inherently minimal. Therefore, the case of wireless networks with preconfigured transmission power levels at their nodes is a practical issue worth studying.

In this paper we propose an optimal energy efficient broadcasting algorithm, called Optimal Total and Residual Energy Multicost Broadcast (abbreviated OTREMB) algorithm, for wireless networks consisting of nodes with preconfigured levels of transmission power. Our algorithm is optimal, in the sense that it can optimize any desired function of the total power consumed by the broadcasting task and the minimum of the current residual energies of the nodes, provided that the optimization function is monotonic in each of these parameters. The proposed algorithm takes into account these two energy-related parameters in selecting the optimal sequence of nodes for performing the broadcast, but it has non-polynomial complexity. We also present a relaxation of the optimal algorithm, to be referred to as the Near-Optimal Total and Residual Energy Multicost Broadcast (abbreviated NOTREMB) algorithm, that produces a near-optimal solution to the energy-efficient broadcasting problem in polynomial time.

The performance of the proposed algorithms is evaluated through simulations. We compare the optimal and near-optimal algorithms to other representative algorithms for energy-efficient broadcasting. Our results show that the proposed algorithms outperform the other algorithms by making better use of the network energy reserves. Another important result is that the near-optimal algorithm performs comparably to the optimal algorithm, at a significantly lower computation cost.

The remainder of the paper is organized as follows. In Section 2 we discuss prior related work. In Sections 3 and 4 we present the optimal and near-optimal algorithms introduced in this paper for energy-efficient broadcasting. In Section 5 the simulations setting is outlined and the performance results are presented. Finally, in Section 6 we give the conclusions drawn from our work.

2 Related Work

Energy-efficiency in all types of communication tasks (unicast, multicast, broadcast) has been considered from the perspective of either minimizing the total energy consumption or maximizing the network lifetime. Most versions of both optimization problems are NP-hard [123]. Two surveys summarizing much of the related work in the field can be found in [45].

A major class of works in the field start with an empty solution which is gradually augmented to a broadcast tree. A seminal work presenting a series of basic energy-efficient broadcasting algorithms, like Minimum Spanning Tree, Shortest Path Tree and Broadcast Incremental Power (BIP), is [7]. The BIP algorithm maintains a single tree rooted at the source node, and new nodes are added to the tree, one by one, on a minimum incremental cost basis. In Broadcast Average Incremental Power (BAIP) algorithm [8] many new nodes can be added at the same step with the average incremental cost defined as the ratio of the minimum additional power required by a node in the current tree to reach these new nodes to the number of new nodes reached. The Greedy Perimeter

Broadcast Efficiency (GPBE) algorithm [9] uses another greedy decision metric, defined as the number of newly covered nodes reached per unit transmission power. In [10], the Minimum Longest Edge (MLE) and the Minimum Weight Incremental Arborescence (MWIA) algorithms are presented. The MLE first computes a minimum spanning tree using as link costs the required transmission powers and then removes redundant transmissions. In MWIA, a broadcast tree is constructed using as criterion a weighted cost that combines the residual energy and the transmission power of each node. In [11], the Relative Neighborhood Graph (RNG) topology is used for broadcasting. In Local Minimum Spanning Tree (LMST) [12] each node builds a one-hop minimum spanning tree. A link is included in the final graph if it selected in the local MSTs of both its edge nodes. In [13] a localized version of the BIP algorithm is presented.

All the aforementioned works assume adjustable node transmission power. One of the few papers that assumes preconfigured power levels for each node is [14], where two heuristics for the minimum energy broadcast problem are proposed: a greedy one, where the criterion for adding a new node in the tree is the ratio of the expended power over the number of the nodes covered by the transmission, and a node-weighted Steiner tree based algorithm.

Local search algorithms perform a walk on broadcast forwarding structures. The walk starts from an initial broadcast topology obtained by some algorithm and in each step, a local search algorithm moves to a new broadcast topology so that the necessary connectivity properties are maintained. The rule used at each step for selecting the next topology is energy-related and the algorithm terminates when no further improvement can be obtained. In [7], the Sweep heuristic algorithm was proposed to improve the performance of BIP by removing transmissions that are unnecessary, due to the wireless broadcast advantage. Iterative Maximum-Branch Minimization (IMBM) [15] starts with a trivial broadcast tree where the source transmits directly to all other nodes and at each step replaces the longest link with a two-hop path that consumes less energy. In [16], EWMA is proposed that modifies a minimum spanning tree by checking whether increasing a node's power so as to cover a child of one of its children, would lead to power savings. The r -Shrink heuristic [16] is applied to every transmitting node and shrinks its transmission radius so that less than r nodes hear each transmission. The LESS heuristic [17] permits a slight increase in the transmission power of a node so that multiple other nodes can stop transmitting or reduce their transmission power.

3 The Optimal Total and Residual Energy Multicost Broadcast Algorithm

The objective of the Optimal Total and Residual Energy Multicost Broadcasting (OTREMB) algorithm is to find, for a given source node, an optimal sequence of nodes for transmitting, so as to implement broadcasting in an energy-efficient way. In particular, it selects a transmission schedule that optimizes any

desired function of the total power T consumed by the broadcasting task and the minimum R of the residual energies of the nodes, provided that the optimization function used is monotonic in each of these parameters, T and R .

The OTREMB algorithm's operation consists of two phases, in accordance with the general multicost algorithm [18] on which it is based. In the first phase, the source node u calculates a set of candidate node transmission sequences \mathcal{S}_u , called set of non-dominated schedules, which can send to all nodes any packet originating at that source. In the second phase, the optimal sequence of nodes for broadcasting is selected based on the desired optimization function..

3.1 The Enumeration of the Candidate Broadcast Schedules

In the first phase of the OTREMB algorithm, every source node u maintains at each time a set of candidate broadcast schedules \mathcal{S}_u . A broadcast schedule $S \in \mathcal{S}_u$ is defined as $S = \{(u_1 = u, u_2, \dots, u_h), V_S\}$, where (u_1, u_2, \dots, u_h) is the ordered sequence of nodes used for transmission and $V_S = (R_S, T_S, P_S)$ is the cost vector of the schedule, consisting of: the minimum residual energy R_S of the sequence of nodes u_1, u_2, \dots, u_h , the total power consumption T_S caused when these nodes are used for transmission and the set P_S of network nodes covered when nodes u_1, u_2, \dots, u_h transmit a packet.

When node u_i transmits a packet at distance r_i , the energy expended is taken to be proportional to r_i^a , where a is a parameter that takes values between 2 and 4. Because of the broadcast nature of the medium and assuming omni-directional antennas, a packet being sent or forwarded by a node can be correctly received by any node within range r_i of the transmitting node u_i . Therefore, broadcast communication tasks in these networks correspond to finding a sequence of transmitting nodes, instead of a sequence of links as it is common in the wire-line world. The assumption of omni-directional antennas is not necessary for the proposed algorithms to work, provided that we know the set of nodes $D(u_i)$ that can correctly decode a packet transmitted by node u_i ; the performance results to be presented in Section 5, however, assume that omni-directional antennas are used.

In general, the routing process, regardless of whether unicasting, multicasting or broadcasting is performed, involves two levels: the information exchange level and the algorithmic level. Information exchange protocols deal with collecting and disseminating network state information, while algorithms calculate the optimal way to perform the desired communication task using this information. Our focus is on the broadcast algorithmic level and do not consider issues regarding the collection of the network information. This way we give lower bounds on the energy efficiency of the proposed solutions.

Initially, each source node u has only one broadcast schedule $\{\emptyset, (\infty, 0, u)\}$, with no nodes, infinite node residual energy, zero total power consumption, while the set of covered nodes contains only the source. The candidate broadcast schedules from source node u are calculated as follows:

1. Each broadcast schedule $S = \{(u_1, u_2, \dots, u_{i-1}), (R_S, T_S, P_S)\}$ in the set of non-dominated schedules \mathcal{S}_u is extended, by adding to its sequence of transmitting nodes a node $u_i \in P_S$ that can transmit to some node u_j not contained in P_S . If no such nodes u_i and u_j exist, we proceed to the final step.

Then the schedule S is used to obtain an extended schedule S' as follows:

- node u_i is added to the sequence u_1, u_2, \dots, u_{i-1} of transmitting nodes
- $R_{S'} = \min(R_i, R_S)$, where R_i is the residual energy of node u_i
- $T_{S'} = T_S + T_i$, where T_i is the (fixed) transmission power of node u_i
- the set of nodes $D(u_i)$ that are within transmission range from u_i are added to the set P_S .
- the extended schedule

$S' = \{(u_1, \dots, u_{i-1}, u_i), (\min(R_S, R_i), T_S + T_i, P_S \cup D(u_i))\}$ obtained in the way described above is added to the set \mathcal{S}_u of candidate schedules.

2. Next, a *domination relation* between the various broadcast schedules of source node u is applied, and the schedules found to be dominated are discarded. In particular, a schedule S_1 is said to *dominate* a schedule S_2 when $T_1 < T_2$, $R_1 > R_2$ and $P_1 \supset P_2$. In other words schedule S_1 dominates schedule S_2 if it covers a superset of nodes than the one covered by S_2 , using less total transmission power and with larger minimum residual energy on the nodes it uses. All the schedules found to be dominated by another schedule are discarded from the set \mathcal{S}_u .
3. The procedure is repeated, starting from the first step 1, for all broadcast schedules in \mathcal{S}_u that meet the above conditions. If no schedule $S \in \mathcal{S}_u$ can be extended further, we go to the final step.
4. Among the schedules in \mathcal{S}_u we form the subset of schedules S for which P_S includes all network nodes. This subset is called the *set of non-dominated schedules* for broadcasting from source node u , and is denoted by $\mathcal{S}_{u,\mathcal{B}}$.

Note that the node residual energy is a restrictive cost parameter, since its minimum value on the nodes used by a schedule defines the schedule's residual energy, while the node transmission power is an additive cost parameter. This is because the residual energy on a set of nodes is more accurately characterized by its minimum value among all nodes in the set, while the power consumed by a set of nodes is described by the sum of their transmission powers.

3.2 The Selection of the Optimal Broadcast Schedule

In the second phase of the OTREMB algorithm, an optimization function $f(V_S)$ is applied to the cost vector V_S of every non-dominated schedule $S \in \mathcal{S}_{u,\mathcal{B}}$ of source node u , produced in the first phase. The optimization function combines the cost vector parameters to produce a scalar metric representing the cost of using the corresponding sequence of nodes for broadcasting. The schedule with the minimum cost is selected. In the performance results described in Section 5, the optimization function used is

$$f(S) = \frac{T_S}{R_S}, \text{ for } S \in \mathcal{S}_{u,\mathcal{B}},$$

which favors, among the schedules that cover all nodes, those that consume less total energy T_S and whose residual energy R_S is larger.

Other optimization functions could also be used, depending on the interests of the network. Also, our algorithm, with straight-forward modifications, can include parameters other than the transmission power and the residual energies of the nodes. For example, we can include as a cost parameter the number h_S of transmissions required by schedule S to complete the broadcast (which is also related to the broadcast delay, in the absence of other traffic and queueing delays in the network), and the optimization function used could also incorporate this parameter when deciding the optimal schedule. The only requirement is that the optimization function has to be monotonic in each of its parameters (e.g., an increasing function of T_S and h_S , a decreasing function of R_S , etc).

Theorem 1. *If the optimization function $f(V_S)$ is monotonic in each of the parameters involved, the OTREMB algorithm finds the optimal broadcast schedule.*

Proof. Since $f(V_S)$ is monotonic in each of its parameters, the optimal schedule has to belong to the set of non-dominated schedules (a schedule S_1 that is dominated by a schedule S_2 , meaning that it is worse than S_2 with respect to all the parameters, cannot optimize f). Therefore, it is enough to show that the set \mathcal{S}_u computed in Steps 1-3 of OTREMB includes all the non-dominated schedules for broadcasting from node u .

We let $S = ((u_1, u_2, \dots, u_h), (R_S, T_S, P_S))$ be a non-dominated schedule that has minimal number of transmissions h among the schedules not produced by OTREMB. Then for the schedule $S' = ((u_1, u_2, \dots, u_{h-1}), (R_{S'}, T_{S'}, P_{S'}))$ we have that $R_S = \min(R_{S'}, R_h)$, $T_S = T_{S'} + T_h$, and $P_S = P_{S'} \cup D(u_h)$. The fact that S is non-dominated and was not produced by OTREMB, implies that S' was not produced by OTREMB either. Since S is a non-dominated schedule with minimal number of transmissions among those not produced by OTREMB, and S' was not produced by OTREMB and uses less transmissions, this means that S' is dominated. However, since S is non-dominated, this means that S' is also non-dominated (otherwise, the schedule S'' that dominates S' , in the sense that it has $T_{S''} < T_{S'}$, $R_{S''} > R_{S'}$ and $P_{S''} \supset P_{S'}$, extended by the transmission from node u_h would dominate S), which is a contradiction. \square

The OTREMB algorithm solves the energy-efficient broadcasting problem optimally and has non-polynomial complexity. The complexity of the OTREMB multicost algorithm is related to the number of different non-dominated schedules S produced by the first phase of the algorithm. This number increases exponentially to the number n of wireless nodes, since the set of covered nodes P_S can take 2^n different values. Moreover, based on [6] the complexity of any multicost algorithm using one additive (T) and one restrictive (R) parameter is polynomial. So, there are cases where, in the second phase of the OTREMB algorithm, the optimization function f is applied to an exponential large number of schedules S . This leads to exponential execution time and to non-polynomial worst case complexity.

4 The Near-Optimal Total and Residual Energy Multicost Broadcast Algorithm

The OTREMB algorithm finds the schedule that optimizes the desired optimization function $f(V_S)$, but it has non-polynomial complexity, since the number of non-dominated schedules generated by the first phase of the algorithm can be exponential. In order to obtain a polynomial time algorithm, we relax the domination condition so as to obtain a smaller number of candidate schedules. In particular, we define a *pseudo-domination* relation among schedules, according to which a schedule S_1 *pseudo-dominates* schedule S_2 , if $T_1 < T_2$, $R_1 > R_2$, and $|P_1| > |P_2|$, where T_i , R_i , $|P_i|$ are the total transmission power, the residual energy of the broadcast nodes and the cardinality of the set of nodes covered by schedule S_i , $i = 1, 2$, respectively. When this pseudo-domination relationship is used in step 2 of the OTREMB algorithm, it results in more schedules being pruned (not considered further) and smaller algorithmic complexity. In fact, by weakening the definition of the domination relationship the complexity of the algorithm becomes polynomial (this can easily be shown by arguing that T_i , R_i and $|P_i|$ can take a finite number of values, namely, at most as many as the number of nodes). The decrease in time complexity, however, comes at the price of losing the optimality of the solution. We will refer to this this near-optimal variation of the OTREMB algorithm, as the Near-Optimal Total and Residual Energy Multicost Broadcast algorithm (abbreviated NOTREMB).

The set of covered nodes P_S can take n different values, where n is the number of wireless nodes. As a result the number of different non-dominated schedules S increases polynomially to n , leading to a polynomial complexity for the NOTREMB algorithm.

5 Performance Results

5.1 Simulation Setting

We implemented and evaluated the proposed algorithms, using the Network Simulator ns-2 [19]. We use a 4×4 two-dimensional grid network topology of 16 stationary nodes with distance of 50 meters between neighboring nodes. Each node's transmission radius is fixed at a value uniformly distributed between 50 and 100 meters.

The performance of the OTREMB and NOTREMB algorithms is evaluated in comparison to established solutions for energy-efficient broadcasting like the BIP algorithm [7], the MWIA algorithm [10] that constructs a minimum spanning tree using as link cost the ratio of the transmission power over the residual energy of the transmitting node, and the BAIP heuristic, which uses as criterion for the addition of a node in the tree the power consumed when using the corresponding link over the number of newly covered nodes. The BAIP heuristic corresponds to the Greedy-h heuristic [14], which, to the best of our knowledge, is the best solution proposed so far for energy-efficient broadcasting when the nodes' transmission power is fixed.

The broadcasting strategies are evaluated under the packet evacuation model, where each node starts with a certain amount of initial energy and a given number of packets to be broadcasted. In our experiments the initial energy E_0 is taken to be equal for all nodes (5, 10 and 100 Joules). Each node broadcasts 200-1000 packets (at steps of 200).

In addition to the evacuation model, we also evaluate the proposed algorithms under the infinite time horizon model. In this model, the time axis is divided into rounds of a dynamic duration. At the beginning of each round the node energy levels are restored to a certain value and a constant number of additional packets to be broadcasted is inserted in every node, to be routed together with any packets whose broadcast was not completed in the previous rounds. Each round terminates (and a new round begins) when the residual energy of at least half of the network nodes falls below a limit. This way nodes are prevented from running out of energy and thus dropping the packets stored in their queues.

5.2 Packet Evacuation Model

In the packet evacuation model, each node starts with a certain amount of initial energy and a given number of packets to be broadcasted. We study the performance of the aforementioned algorithms until all packets are successfully broadcasted or until no more transmission can take place to the lack of energy reserves.

As far as the average number of transmissions h required to complete a broadcast is concerned (Figure 1a), OTREMB outperforms all the other algorithms, with NOTREMB achieving only slightly larger h . BAIP also seems to perform similarly to OTREMB and NOTREMB, but this is rather misleading, since as it will be shown below (see Figure 2) BAIP does not successfully complete the same number of broadcasts with these schemes.

The overall energy expenditure of the algorithms is depicted in Figure 1b, where the node average residual energy R at the end of the experiment is shown, for the case of initial energy equal to 5 Joules. Generally, the energy reserves of the nodes decrease as the number of packets broadcasted increases. In this case that the nodes have finite energy reserves, R stops decreasing when a certain

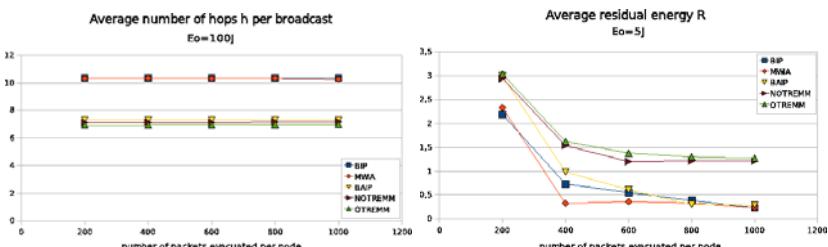


Fig. 1. (a) The average number of transmissions h per broadcast for all the algorithms evaluated when the nodes initial energy is equal to 100 Joules. (b) The average residual energy R at the end of the experiment for all the algorithms evaluated when the nodes initial energy is equal to 5 Joules.

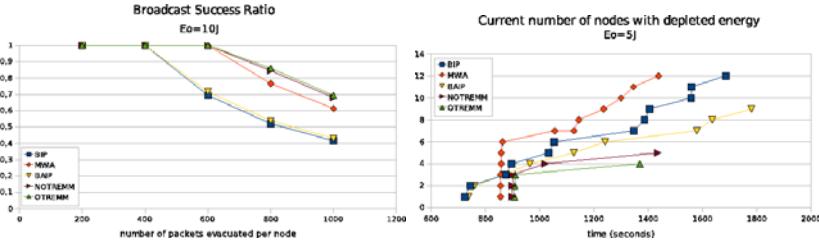


Fig. 2. (a) The broadcast success ratio p for all the algorithms evaluated when the nodes initial energy is equal to 10 Joules, (b) the current number of nodes L with depleted energy for all the algorithms evaluated when the nodes initial energy is equal to 5 Joules

number of packets (400-600 packets per node, depending on the scheme used) are broadcasted in the network. This happens because, beyond this point, many nodes run out of energy and the network gets disconnected. As a result, regardless of the increase in the incoming traffic no more packet transmissions can take place and energy consumption does not grow further. The OTREMB and NOTREMB algorithms utilize the node energy reserves more efficiently, compared to the rest of the algorithms, resulting in a higher residual energy R at the end of an evacuation period. Even though OTREMB selects the most energy-efficient set of nodes for broadcasting, NOTREMB achieves comparable results. This strengthens our belief that the NOTREMB algorithm may be preferable in practice to the OTREMB algorithm, obtaining most of the performance benefits of the optimal algorithm at a smaller computational cost.

Figure 2a depicts the broadcast success ratio p for all the algorithms considered. Clearly, OTREMB, with NOTREMB being only slightly inferior, outperforms all the other algorithms. The performance of the BIP and BAIP algorithms starts degrading even for small traffic load, and only the MWIA algorithm stays close to the OTREMB and NOTREMB algorithms. The characteristic enabling MWIA to also perform rather well is that its selection criterion is dynamic, since it involves the time-varying current residual energy of the nodes. In contrast, the BIP and BAIP algorithms do not change their broadcast paths and, therefore, they quickly exhaust the node energy reserves. The OTREMB and NOTREMB algorithms spread traffic more uniformly across the network, with nodes remaining operational and able to broadcast packets for longer times, as the following results will also indicate.

The longer network lifetime achieved by the OTREMB and NOTREMB algorithms compared to the other broadcast algorithms can be observed in Figure 2b, where the current number of nodes L with depleted energy reserves is presented as a function of time. The OTREMB and NOTREMB algorithms clearly result in fewer nodes running out of energy compared to the other algorithms. Furthermore, these nodes run out of energy later than they do in the other algorithms. The BIP algorithm seems to have the worst performance. MWIA manages to spread energy consumption uniformly across a subset of network nodes, and

therefore when these nodes run out of energy, they do so almost simultaneously. The BAIP algorithm performs better than MWIA and BIP, but significantly worse than OTREMB and NOTREMB with respect to network lifetime.

5.3 Infinite Time Horizon Model

In the infinite time horizon setting, the broadcasting strategies are evaluated assuming packets and energy are generated over an infinite time horizon, according to a round-based scenario. At the beginning of each round, the node energy reserves are restored to a certain level, and an equal number of packets N to be broadcasted is generated at every node. A round terminates when the residual energy of at least half of the network nodes falls below a certain safety limit. Packets that are not successfully broadcasted during a round, continue from the point they stopped (e.g., a node with residual energy levels below the safety limit) in the following round(s) until their broadcast is completed. The succession of rounds continues until the network reaches steady-state, or until it becomes inoperable (unstable). We evaluate the algorithms using the broadcast success ratio p and the average broadcast delay D as performance metrics.

Figure 3a presents the broadcast success ratio p at steady state, for a different number of broadcast packets N inserted at each node per round. We observe that even for relatively light traffic inserted in each round, the BIP, the MWIA and the BAIP algorithms are not able to successfully broadcast all the packets generated. In particular, the BIP scheme remains stable for load up to $N = 15$ packets per node per round, and the BAIP scheme for load up to $N = 17$ packets per node per round. The MWIA scheme performs slightly better, remaining stable for up to $N = 19$ packets per node per round. The OTREMB and NOTREMB schemes have the maximum stability region (maximum broadcast throughput) and remain stable for up to $N = 21$ packets per node per round. By taking into account energy-related cost parameters and switching through multiple energy-efficient paths, both OTREMB and NOTREMB spread energy consumption more evenly and increase the volume of broadcast traffic that can be successfully served. The NOTREMB algorithm performs comparably to the OTREMB algorithm, and only for the heavier loads of incoming traffic its success ratio p falls significantly below that of the optimal algorithm. This is important, considering the great gain in computational effort achieved when using the NOTREMB instead of the OTREMB algorithm.

Figure 3b shows the average broadcast delay D , measured in packet times, at steady state for all the algorithms evaluated, as a function of the number of broadcast packets inserted in the network per node and per round. Recall that the broadcast delay D includes the delays incurred by a packet during all the rounds that elapse from the time it is generated until the time it reaches all the nodes in the network. As it can be seen from Figure 3b, the delay versus traffic load curves of the BIP, MWIA and BAIP algorithms are above those of the OTREMB and the NOTREMB algorithms. Since packets whose broadcast is not completed during a round fill the node queues and congest the network, the average delay of the BIP, MWIA and BAIP algorithms quickly becomes very large. Naturally, when the traffic load inserted increases beyond each scheme's

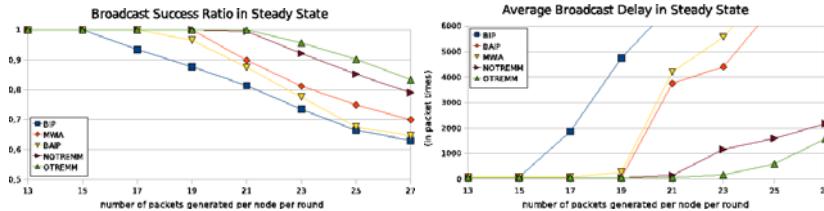


Fig. 3. (a) The broadcast success ratio p , and (b) the average broadcast delay D in the steady-state of the algorithms evaluated, for a different number of broadcast packets N inserted in the network

maximum stable throughput, the delays will also become unbounded, and the success ratio p will start falling. The OTREMB and the NOTREMB algorithms have smaller average delay D and remain stable for higher loads than the other schemes considered. Again, the NOTREMB algorithm manages to achieve similar performance to that of the OTREMB algorithm, and only at the end its performance deteriorates.

6 Conclusions

We studied energy-aware broadcasting in wireless networks, and proposed an optimal (OTREMB) and a near-optimal (NOTREMB) algorithm, based on the multicost concept. We evaluated the performance of the proposed algorithms and compared it to that of established heuristics. Our results show that the proposed multicost algorithms outperform the other heuristic algorithms considered, consuming less energy and successfully broadcasting more packets to their destination, under both the packet evacuation and the infinite time horizon model. An interesting conclusion drawn from our simulations is that the near-optimal multicost algorithm, NOTREMB, has similar performance to that of the optimal multicost algorithm, OTREMB, while having considerably smaller execution time.

Acknowledgments

C. Papageorgiou was supported by GSRT through PENED project 03EΔ207, funded 75% by the EC and 25% by the Greek State and the private sector.

References

1. Čagalj, M., Hubaux, J.P., Enz, C.: Minimum-energy broadcast in all-wireless networks: Np-completeness and distribution issues. In: Proc. of the 8th Ann. Int'l Conf. on Mobile Computing and Networking. MobiCom 2002, pp. 172–182. ACM, New York (2002)
2. Li, D., Liu, Q., Hu, X., Jia, X.: Energy efficient multicast routing in ad hoc wireless networks. ComCom 30(18), 3746–3756 (2007)

3. Liang, W.: Constructing minimum-energy broadcast trees in wireless ad hoc networks. In: Proc. of the 3rd ACM Int'l Symp. on Mobile Ad Hoc Networking & computing. MobiHoc 2002, pp. 112–122. ACM, New York (2002)
4. Guo, S., Yang, O.W.W.: Energy-aware multicasting in wireless ad hoc networks: A survey and discussion. *Computer Communications* 30(9), 2129–2148 (2007)
5. Athanassopoulos, S., Caragiannis, I., Kaklamanis, C., Kanellopoulos, P.: Experimental comparison of algorithms for energy-efficient multicasting in ad hoc networks. In: ADHOC-NOW, pp. 183–196 (2004)
6. Wang, Z., Crowcroft, J.: Quality-of-Service Routing for Supporting Multimedia Applications. *J. of Selected Areas in Communications* 14(7), 1228–1234 (1996)
7. Wieselthier, J., Nguyen, G., Ephremides, A.: On the construction of energy-efficient broadcast and multicast trees in wireless networks. In: Proceedings IEEE INFOCOM, pp. 585–594 (2000)
8. Wan, P.J., Calinescu, G., Li, X., Frieder, O.: Minimum-energy broadcast routing in static ad hoc wireless networks. In: Proceedings IEEE INFOCOM, vol. 2, pp. 1162–1171 (2001)
9. Kang, I., Poovendran, R.: A novel power-efficient broadcast routing algorithm exploiting broadcast efficiency. In: IEEE 58th Vehicular Technology Conf., 2003, vol. 5, pp. 2926–2930 (2003)
10. Cheng, M., Sun, J., Min, M., Li, Y., Wu, W.: Energy-efficient broadcast and multicast routing in multihop ad hoc wireless networks. *Wireless Communications and Mobile Computing* 6(2), 213–223 (2006)
11. Cartigny, J., Simplot, D., Stojmenovic, I.: Localized minimum-energy broadcasting in ad-hoc networks. In: Proceedings IEEE INFOCOM (2003)
12. Li, X., Demmel, J., Bailey, D., Henry, G., Hida, Y., Iskandar, J., Kahan, W., Kapur, A., Martin, M.C., Tung, T., Yoo, D.J.: Design, implementation and testing of extended and mixed precision BLAS. *ACM Trans. on Mathematical Software* (2002)
13. Ingelrest, F., Simplot-Ryl, D.: Localized broadcast incremental power protocol for wireless ad hoc networks. *Wireless Networks* 14(3), 309–319 (2008)
14. Li, D., Jia, X., Liu, H.: Energy efficient broadcast routing in static ad hoc wireless networks. *IEEE Trans. on Mobile Computing* 3(2), 144–151 (2004)
15. Li, F., Nikolaidis, I.: Proc. of the 26th Ann. Conf. on Local Computer Networks (2001)
16. Das, A., Marks, R., El-Sharkawi, M., Arabshahi, P., Gray, A.: r-shrink: A heuristic for improving minimum power broadcast trees in wireless networks. In: Global Telecommunications Conference. GLOBECOM 2003, vol. 1. IEEE, Los Alamitos (2003)
17. Kang, I., Poovendran, R.: Broadcast with heterogeneous node capability. In: Global Telecommunications Conference, 2004. GLOBECOM 2004, vol. 6, pp. 4114–4119. IEEE, Los Alamitos (2004)
18. Gutierrez, F., Varvarigos, E., Vassiliadis, S.: Multi-cost routing in max-min fair share networks. In: Proc. 38th Ann. Allerton Conf. on Communication, Control and Computing, vol. 2, pp. 1294–1304 (2000)
19. The Network Simulator NS-2, <http://www.isi.edu/nsnam/ns/>

Author Index

- Abramson, David 389
Acacio, Manuel E. 900
Alam, S.R. 334
Alcover, Rosa 1040
Aldinucci, Marco 678
Alfaro, Francisco J. 1089
Amestoy, Patrick 719
Antoniu, Gabriel 404
Aragón, Juan L. 321
Arbenz, Peter 719
Augonnet, Cédric 863
Augustin, Werner 772
Ayguadé, Eduard 837, 851
Aykanat, Cevdet 375
- Backes, Werner 960
Badia, Rosa M. 851
Bal, Henri 3
Barrett, R.F. 334
Baude, Françoise 691
Baumes, Laurent 974
Benavides, José Ignacio 924
Benoit, Anne 253
Bergdorf, Michael 721
Bermúdez, Aurelio 1029
Berthold, Jost 990
Best, Micah J. 912
Bhatelé, Abhinav 1015
Bic, Lubomir F. 590
Bisseling, Rob 989
Bodin, François 837
Boeres, Cristina 191
Bohm, Eric 1015
Bougé, Luc 404
Bouziane, Hinde Lilia 678
Brown, Aron 912
Brown, J. 747
Brownsworth, Andrew 912
Bubak, Marian 95
Burckhardt, K. 747
Buyya, Rajkumar 417
- Cappello, Franck 466
Caron, Eddy 602
Carrington, Laura 135
- Carzaniga, Antonio 511
Casado, Rafael 1029
Cavé, Vincent 691
Chapman, Barbara 837
Chen, Jian 361
Chen, Wenguang 81
Chi, Chi-Hung 442
Chirivella, Vicente 1040
Chung, I-Hsin 33
Clauss, Philippe 974
Claver, José M. 1089
Collet, Pierre 974
Cong, Guogjing 33
Corma, Avelino 974
Costa, Paolo 561
Coti, Camille 466
Couture-Beil, Alex 912
Cunningham, Christina 454
- D’Ambra, Pasqua 719
Danelutto, Marco 678
Darte, Alain 295
Datta, Ajoy K. 602
Depardon, Benjamin 602
DeRose, Luiz 7
Develder, Chris 1013
Devi, UmaMaheswari C. 265
Diakité, Sékou 203
Dias, Ricardo J. 349
Dickie, Ryan 912
Dieterle, Mischa 990
Dillencourt, Michael B. 590
Diniz, Pedro C. 295
Doallo, Ramón 630
Donath, Stefan 735
Duato, José 1040
Dursun, Hikmet 642
- Eckart, Benjamin 57
Eertink, Henk 1103
- Fahringer, Thomas 95
Fan, Dongrui 110, 948
Fan, Lingyuan 361

- Fedorova, Alexandra 912
 Feichtinger, Christian 735
 Fernández, Juan 900
 Flich, José 1040
 Foran, David J. 936
 Franco, Daniel 1078
- Gan, Ge 839
 Gao, Guang R. 839
 Gaona, Epifanio 900
 Garbacki, Paweł 481
 García, José M. 321
 García-López, Pedro 535
 Geijn, Robert van de 809
 Genius, Daniela 216
 Georgiadis, Giorgos 483
 Ghazali, Nasser 912
 Gijzen, Martin van 719
 Gillen, Daniel 590
 Gkantsidis, Christos 481
 Goglin, Brice 1065
 Gómez, Andrés 630
 Gómez-Luna, Juan 924
 Gong, Hao 361
 Gong, Leiguang 936
 González, Antonio 297
 González-Domínguez, Jorge 630
 González-Linares, José María 924
 Gorlatch, Sergei 654
 Götz, Jan 735
 Graham, Peter 69
 Grossman, Max 887
 Grosu, Daniel 165
 Guil, Nicolás 924
- Ha, Soonhoi 7
 Hall, Cyrus 511
 Harmer, Terence 454
 He, Xubin 57
 Hejazialhosseini, Babak 721
 Herault, Thomas 466
 Herrera, Blas 535
 Heuveline, Vincent 772
 Hiemstra, Djoerd 347
 Hollingsworth, Jeffrey K. 21
 Huang, Zhi Feng 912
 Huet, Fabrice 629
 Humphrey, Marty 389
- Iamnitchi, Adriana 481
 Ibáñez, Pablo 149
 Iglberger, Klaus 760
 Igual, Francisco D. 851
 Iosup, Alexandru 95, 390
 Iwasaki, Hideya 666
- Jagode, H. 334
 Jeannot, Emmanuel 165
 Jelasity, Márk 523
 Jha, Shantenu 629
 Jin, Chao 417
 Jukan, Admela 1013
 Junqueira, Flavio 589
 Juurlink, Ben 295
- Kalé, Laxmikant V. 1015
 Kalia, Rajiv K. 642
 Kandemir, Mahmut 122
 Karatza, Helen 165
 Karl, Wolfgang 295
 Kegel, Philipp 654
 Kemme, Bettina 178
 Kemper, Alfons 347
 Kerbyson, Darren J. 166
 Kermarrec, A.-M. 574
 Kielmann, Thilo 7
 Kienhuis, Bart 837
 Klepaci, David 33
 Kluge, Michael 45
 Knüpfer, Andreas 45
 Kokkinos, Panagiotis C. 1104
 Korch, Matthias 785
 Kortuem, Gerd 1103
 Kostić, Dejan 589
 Koumoutsakos, Petros 721
 Kounev, Samuel 97
 Kuehn, J.A. 334
- Laat, Cees de 1013
 Labarta, Jesús 851
 Lachiche, Nicolas 974
 Lalis, Spyros 228
 Lampsas, Petros 228
 Lander, Arthur D. 590
 Lanteri, Stéphane 691
 Larmore, Lawrence L. 602
 Laurenzano, Michael A. 135
 Lebhar, Emmanuelle 989
 Liang, Liang 361

- Lira, Javier 297
 Liu, Fangbin 821
 Liu, Jiming 590
 Liu, Y. 574
 Llabería, J.M. 149
 Long, Guoping 110
 Loogen, Rita 990
 López, P. 309
 Loukopoulos, Thanasis 228
 Lourenço, João M. 349
 Lu, Qingming 361
 Lugones, Diego 1078
 Lupu, Mihai 498
 Luque, Emilio 1078
 Lysne, Olav 1052
- Maassen, Jason 629
 Maitre, Ogier 974
 Mambretti, Joe 1013
 Manguoglu, Murat 797
 Manzano, Joseph 839
 Marchal, Loris 203
 Marqués, Mercedes 809
 Martín, María J. 630
 Martínez, Raúl 1089
 Mathias, Elton 691
 Mayo, Rafael 851
 Mehnert-Spahn, John 429
 Merrer, E. Le 574
 Michiardi, Pietro 548
 Mizuno, Ken 704
 Molina, Carlos 297
 Morajko, Anna 7
 Morin, Christine 429, 615
 Moriyama, Takao 33
 Mottishaw, Shane 912
 Mühl, Gero 97
 Müller, Matthias 45
 Munier Kordon, Alix 216
 Muralidhar, K. 747
 Murata, Hiroki 33
 Mustard, Craig 912
- Nagel, Wolfgang E. 45
 Nakaike, Takuya 704
 Nakano, Aiichiro 642
 Nakatani, Toshio 704
 Namyst, Raymond 863
 Nascimento, Aline P. 191
 Negishi, Yasushi 33
- Nicod, Jean-Marc 203
 Nicolae, Bogdan 404
 Nomura, Ken-ichi 642
 Nosher, John L. 936
- Oktay, K. Yasin 375
 Ong, Hong 57
 Osterloh, Andre 1003
 Otto, Frank 875
- Pankratius, Victor 9, 875
 Papageorgiou Christos A. 1104
 Papatriantafilou, Marina 483
 Parzy jegla, Helge 97
 Pecero, Johnatan E. 241
 Peng, Liu 642
 Pérez, Christian 678
 Perrone, Michael 1
 Perrott, Ron 454
 Petit, S. 309
 Philippe, Laurent 203
 Pierre, Guillaume 442, 561, 589
 Pietracaprina, Andrea 989
 Pietzuch, Peter R. 589
 Pimentel, Andy D. 7
 Pineau, Jean-François 281
 Plata, Oscar 837
 Pohl, Thomas 735
 Poole, S.W. 334
 Prehofer, Christian 1103
 Prieto, Manuel 347
- Quintana-Ortí, Enrique S. 809, 851
 Quintana-Ortí, Gregorio 809
- Raghavan, Padma 122
 Rai, Vivek 561
 Rauber, Thomas 785
 Rebello, Vinod E.F. 191
 Richling, Jan 97
 Ripeanu, Matei 95
 Robert, Yves 281
 Robles-Gómez, Antonio 1029
 Ropars, Thomas 429, 615
 Rossinelli, Diego 721
 Rowstron, Antony 4
 Rüde, Ulrich 735, 760
 Rutar, Nick 21
- Sahelices, Benjamín 149
 Sahuquillo, J. 309

- Sameh, Ahmed H. 797
 Sánchez, Daniel 321
 Sánchez, José L. 1089
 Sánchez-Artigas, Marc 535
 Sancho, José Carlos 166
 Sankaran, R. 334
 Sarkar, Vivek 887
 Schaefer, Christoph A. 9
 Schellmann, Maraike 654
 Schenk, Olaf 797
 Schoettner, Michael 429
 Schröter, Arnd 97
 Scott, Stephen L. 57
 Seinstra, Frank J. 821
 Sena, Alexandre 191
 Seymour, Richard 642
 Shantharam, Manu 122
 Shi, Guangyu 361
 Simon, G. 574
 Singh, Rajendra 69
 Skeie, Tor 1052
 Slot, Marco 561
 Snavely, Allan 135
 Solheim, Åshild Grønstad 1052
 Stotzer, Eric 837
 Strohbach, Martin 1103
 Sun, Xian-He 95
 Székely, G. 747
 Szalay, Alex 347
 Szczerba, D. 747
 Taboada, Guillermo L. 630
 Tagliasacchi, Andrea 912
 Talia, Domenico 629
 Tan, Guangming 948
 Tanno, Haruto 666
 Thibault, Samuel 863
 Tichy, Walter F. 9, 875
 Tikir, Mustafa M. 135
 Tiskin, Alexander 989
 Toka, Laszlo 548
 Tölgyesi, Norbert 523
 Touriño, Juan 630
 Truong, Hong-Linh 95
 Trystram, Denis 241
 Turk, Ata 375
 Tziritas, Nikos 228
 Ubal, R. 309
 Varvarigos, Emmanouel A. 1104
 Vashishta, Priya 642
 Viñals, Víctor 149
 Vivien, Frédéric 281
 Voulgaris, Spyros 481
 Vu, Quang Hieu 498
 Wacrenier, Pierre-André 863
 Wang, Weiqiang 642
 Wang, Xu 839
 Wei, Zhou 442
 Weiss, Jan-Philipp 772
 Weissman, Jon 389
 Wen, Huifang 33
 Wetzel, Susanne 960
 Wolf, Felix 7
 Wolters, Lex 389
 Wright, Peter 454
 Wu, Huaigu 178
 Wu, Sai 498
 Xu, Xiaoyuan 912
 Xue, Haiqiang 361
 Yahyapour, Ramin 165
 Yan, Yonghong 887
 Yang, Lin 936
 Yuan, Nan 948
 Zarza, Gonzalo 1078
 Zhai, Jidong 81
 Zhang, Hong 936
 Zhang, Jin 81
 Zhang, Junchao 110, 948
 Zhao, Ben 481
 Zheng, Weimin 81
 Zhou, Yongbin 948
 Zine el Abidine, Khouloud 216
 Zomaya, Albert Y. 241