

**Julian Martin Kunkel
Thomas Ludwig
Hans Werner Meuer (Eds.)**

LNCS 8488

Supercomputing

**29th International Conference, ISC 2014
Leipzig, Germany, June 22–26, 2014
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

TU Dortmund University, Germany

Demetri Terzopoulos

University of California, Los Angeles, CA, USA

Doug Tygar

University of California, Berkeley, CA, USA

Gerhard Weikum

Max Planck Institute for Informatics, Saarbruecken, Germany

Julian Martin Kunkel Thomas Ludwig
Hans Werner Meuer (Eds.)

Supercomputing

29th International Conference, ISC 2014
Leipzig, Germany, June 22-26, 2014
Proceedings



Volume Editors

Julian Martin Kunkel
Deutsches Klimarechenzentrum
Bundesstraße 45a, 20146 Hamburg, Germany
E-mail: juliankunkel@googlemail.com

Thomas Ludwig
Deutsches Klimarechenzentrum
Bundesstraße 45a, 20146 Hamburg, Germany
E-mail: ludwig@dkrz.de

Hans Werner Meuer
University of Mannheim, Germany
and Prometheus GmbH
Fliederstraße 2, 74915 Waibstadt, Germany
E-mail: hans.meuer@isc-events.com

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-319-07517-4 e-ISBN 978-3-319-07518-1
DOI 10.1007/978-3-319-07518-1
Springer Cham Heidelberg New York Dordrecht London

Library of Congress Control Number: 2014939416

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

© Springer International Publishing Switzerland 2014

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

protective laws and regulations and therefore free for general use. While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

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

Preface

The International Supercomputing Conference, founded in 1986 as the “Supercomputer Seminar,” has been held annually for the last 29 years. Originally organized by Professor Hans Meuer, Professor of Computer Science at the University of Mannheim and former director of the computer center, the seminar brought together a group of 81 scientists and industrial partners who all shared an interest in high-performance computing. Since then the annual conference has become a major international event within the HPC community, and accompanying its growth in size over the years the conference has moved from Mannheim via Heidelberg, Dresden, and Hamburg to Leipzig. With 2,423 attendees and 153 exhibitors from over 54 countries in 2013, we are optimistic that this steady growth of interest will also have turned ISC’14 into a powerful and memorable event.

In 2007, we decided to strengthen the scientific part of the conference by presenting selected talks on relevant research results within the HPC field. These research paper sessions began as a separate day preceding the conference, where slides and accompanying papers were made available via the conference website. The research paper sessions have since evolved into an integral part of the conference, and this year the scientific presentations were scheduled over four days.

This year, we made several changes to the organization of the research track:

- Full papers and technical short papers with reduced page length can be submitted. This encourages researchers, industry, and data centers to publish latest best-practices and evaluation results.
- Senior reviewers coordinate the discussion among the Program Committee, recommend papers, and compose meta-reviews.
- The bond to the research poster session was tightened: Firstly, the research poster committee is now a subcommittee of the the research paper committee. Secondly, extended abstracts of the posters are included in these proceedings. This year, the research poster committee selected five extended abstracts for publication.
- Digital copies of the ISC research posters are now publicly available for download at: <http://www.isc-events.com/isc14/research-posters.html>.

The call for participation was issued in fall 2013, inviting researchers and developers to submit the latest results of their work to the sessions’ Program Committee. More than 79 short and full papers were submitted from authors all over the world. In a peer-review process, an international committee selected the best 34 papers for publication and for presentation in the research paper sessions.

We are pleased to announce that many fascinating topics in HPC are covered by the proceedings. The papers address the following issues in regards to the development of an environment for exascale supercomputers:

- Scalable applications for 50K+ cores
- Advances in algorithms
- Scientific libraries
- Programming models
- Architectures
- Performance models and analysis
- Automatic performance optimization
- Parallel I/O
- Energy efficiency

We believe that this selection is highly appealing across a number of specializations and therefore the presentations fostered inspiring discussions with the audience.

As in the previous years, two independent award committees selected two papers considered to be of exceptional quality and worthy of special recognition.

- The Gauss Centre for Supercomputing sponsors the Gauss Award. This award is assigned to the most outstanding paper in the field of scalable supercomputing and went to:

Exascale Radio Astronomy: Can We Ride the Technology Wave?
(Erik Vermij, Leandro Fiorin, Christoph Hagleitner, and Koen Bertels)

- PRACE, the Partnership for Advanced Computing in Europe, awards a prize to the best scientific paper by a European student or scientist. This year's award was granted to:

Sustained Petascale Performance of Seismic Simulations with Seis-Sol on SuperMUC (*Alexander Breuer, Alexander Heinecke, Sebastian Rettenberger, Michael Bader, Alice-Agnes Gabriel, and Christian Pelties*)

We would like to express our gratitude to all our colleagues for submitting papers to the ISC scientific sessions, as well as to the members of the Program Committee for organizing this year's appealing program.

The ever-rising number of submissions would have given particular pleasure to Professor Hans Meuer, founder of ISC and friend to us all. Last January, preparations for the scientific part were running full steam when we heard the sad news of his passing.

His death cast a shadow on this year's conference, and he was missed at every booth and every talk. A scientist at heart all his life, it was Hans who fostered the creation of a scientific component for "his" conference series. In but a few years, the trickle of submissions turned into today's sizeable stream and the research paper sessions became an integral part of ISC. Here too, his dream of conjoining science and industry in a single event became true.

Hans Meuer was a perpetual inspiration to the scientific community: Aspiring to ever higher goals, he was observant of new developments and a mentor to a whole generation of scientists. We will continue the ISC scientific track in his memory; as Hans himself was wont to say: “The show must go on.”

June 2014

Julian M. Kunkel
Thomas Ludwig

Organization

Program Committee

Balaji, Pavan	Argonne National Laboratory, USA
Javier Garcia Blas, Francisco	Universidad Carlos III de Madrid, Spain
Bohlouli, Mahdi	University of Siegen, Germany
Brown, Jed	Argonne National Laboratory, USA
Burrows, Eva	University of Bergen, Norway
Cai, Xing	Simula Research Laboratory, Norway
Declat, Damien	Bull, France
Duro, Francisco	Universidad Carlos III de Madrid, Spain
Flouri, Tomas	HITS, Germany
Gross, Lutz	University of Queensland, Australia
Ham, David	Imperial College London, UK
Hannig, Frank	Friedrich-Alexander University Erlangen-Nürnberg, Germany
Haveraaen, Magne	University of Bergen, Norway
Herhut, Stephan	Intel Labs, Santa Clara, USA
Huang, Weicheng	National Applied Research Laboratory, Taiwan
Kindratenko, Volodymyr	National Center for Supercomputing Applications, USA
Koshulko, Oleksiy	Glushkov Institute of Cybernetics NAS, Ukraine
Kunkel, Julian	Deutsches Klimarechenzentrum, Germany
Li, Dong	Oak Ridge National Lab, USA
Fang-Lin, Pang	National Center for High-Performance Computing, Taiwan
Ludwig, Thomas	Deutsches Klimarechenzentrum, Germany
Manjunathaiah, Manjunathaiah	University of Reading, UK
McIntosh-Smith, Simon	University of Bristol, UK
Membarth, Richard	Intel Visual Computing Institute, Saarland University, Germany
Mohr, Bernd	Jülich Supercomputing Center, Germany
Molka, Daniel	ZIH, Technische Universitaet Dresden, Germany
Moskovsky, Alexander	RSC SKIF, Russia

Müller, Matthias	RWTH Aachen University, Germany
Nakajima, Kengo	University of Tokyo, Japan
Nou, Ramon	Barcelona Supercomputing Center, Spain
Olbrich, Stephan	RRZ, Universität Hamburg, Germany
Ortega, Julio	University of Granada, Spain
Phung Huynh, Huynh	A*STAR Institute of High Performance Computing, Singapore
Poulet, Thomas	CSIRO, Australia
Qian, Ying	KAUST, Saudi Arabia
M. Resch, Michael	HLRS Stuttgart, Germany
Rouson, Damian	Center for Computational Earth and Environmental Sciences at Stanford University, USA
Sven-Scholz, Bodo	Heriot-Watt University Edinburgh, UK
Stamatakis, Alexandros	HITS, Germany
Tatebe, Osamu	University of Tsukuba, Japan
Thiyagalingam, Jeyarajan	Oxford University, UK
Tolentino, Matthew	Intel, USA
Tsujita, Yuichi	RIKEN AICS, Japan
Wang, Zhonglei	KIT, Germany
Wild, Thomas	Technische Universität München, Germany

Program Committee for the Research Poster Session

Balaji, Pavan	Moskovsky, Alexander
Bohlouli, Mahdi	Müller, Matthias
Brown, Jed	Nakajima, Kengo
Flouri, Tomas	Nou, Ramon
Gross, Luzt	Olbrich, Stephan
Ham, David	Ortega, Julio
Kindratenko, Volodymyr	Qian, Ying
Koshulko, Oleksiy	Rouson, Damian
Kunkel, Julian	Sven-Scholz, Bodo
Manjunathaiah, Manjunathaiah	Tatebe, Osamu
McIntosh-Smith, Simon	Thiyagalingam, Jeyarajan
Membarth, Richard	Tolentino, Matthew
Mohr, Bernd	Tsujita, Yuichi
Molka, Daniel	Zhonglei, Wang

External Referees

Armour, Wes	Kozlov, Alexey	Shinkarov, Artem
Artiaga, Ernest	Kuhn, Michael	Solnushkin, Konstantin
Bland, Ian	Lu, Huiwei	Stadler, Georg
Chasapis, Konstantinos	Ma, Teng	Stechele, Walter
Dolz, Manuel F.	Martí, Jonathan	Sundar, Hari
Du, Peng	Mian, Lu	Tang, Wai Teng
Duta, Mihai	Mudalige, Gihan	Torres, Raul
Gordon, Stuart	Reiche, Oliver	Zhang, Junchao
Hu, Qi	Röber, Niklas	Zhang, Yongpeng
Hübbe, Nathanael	Sauge, Ludovic	Zheng, Ziming
Koestler, Harald	Schmitt, Christian	Zimmer, Michaela

Table of Contents

Regular Papers

Sustained Petascale Performance of Seismic Simulations with SeisSol on SuperMUC	1
<i>Alexander Breuer, Alexander Heinecke, Sebastian Rettenberger, Michael Bader, Alice-Agnes Gabriel, and Christian Pelties</i>	
SNAP: Strong Scaling High Fidelity Molecular Dynamics Simulations on Leadership-Class Computing Platforms	19
<i>Christian R. Trott, Simon D. Hammond, and Aidan P. Thompson</i>	
Exascale Radio Astronomy: Can We Ride the Technology Wave?	35
<i>Erik Vermij, Leandro Fiorin, Christoph Hagleitner, and Koen Bertels</i>	
On the Performance Portability of Structured Grid Codes on Many-Core Computer Architectures	53
<i>Simon McIntosh-Smith, Michael Boulton, Dan Curran, and James Price</i>	
Performance Predictions of Multilevel Communication Optimal LU and QR Factorizations on Hierarchical Platforms	76
<i>Laura Grigori, Mathias Jacquelin, and Amal Khabou</i>	
Hourglass: A Bandwidth-Driven Performance Model for Sorting Algorithms	93
<i>Doe Hyun Yoon and Fabrizio Petrini</i>	
Performance Analysis of Graph Algorithms on P7IH	109
<i>Xinyu Que, Fabio Checconi, and Fabrizio Petrini</i>	
Sparsifying Synchronization for High-Performance Shared-Memory Sparse Triangular Solver	124
<i>Jongsoo Park, Mikhail Smelyanskiy, Narayanan Sundaram, and Pradeep Dubey</i>	
Scalability and Parallel Execution of OmpSs-OpenCL Tasks on Heterogeneous CPU-GPU Environment	141
<i>Vinoth Krishnan Elangovan, Rosa M. Badia, and Eduard Ayguadé</i>	
Automatic Exploration of Potential Parallelism in Sequential Applications	156
<i>Vladimir Subotic, Eduard Ayguadé, Jesus Labarta, and Mateo Valero</i>	

CoreTSAR: Adaptive Worksharing for Heterogeneous Systems	172
<i>Thomas R.W. Scogland, Wu-chun Feng, Barry Rountree, and Bronis R. de Supinski</i>	
History-Based Predictive Instruction Window Weighting for SMT Processors	187
<i>Gurhan Kucuk, Gamze Uslu, and Cagri Yesil</i>	
The Brand-New Vector Supercomputer, SX-ACE	199
<i>Shintaro Momose, Takashi Hagiwara, Yoko Isobe, and Hiroshi Takahara</i>	
Impact of Future Trends on Exascale Grid and Cloud Computing	215
<i>Ted H. Szymanski</i>	
SADDLE: A Modular Design Automation Framework for Cluster Supercomputers and Data Centres	232
<i>Konstantin S. Solnushkin</i>	
The SIOX Architecture – Coupling Automatic Monitoring and Optimization of Parallel I/O	245
<i>Julian M. Kunkel, Michaela Zimmer, Nathanael Hübbe, Alvaro Aguilera, Holger Mickler, Xuan Wang, Andriy Chut, Thomas Bönisch, Jakob Lüttgau, Roman Michel, and Johann Weging</i>	
Framework and Modular Infrastructure for Automation of Architectural Adaptation and Performance Optimization for HPC Systems	261
<i>Leonardo Fialho and James Browne</i>	
Designing MPI Library with Dynamic Connected Transport (DCT) of InfiniBand: Early Experiences	278
<i>Hari Subramoni, Khaled Hamidouche, Akshey Venkatesh, Sourav Chakraborty, and Dhabaleswar K. Panda</i>	
RADAR: Runtime Asymmetric Data-Access Driven Scientific Data Replication	296
<i>John Jenkins, Xiaocheng Zou, Houjun Tang, Dries Kimpe, Robert Ross, and Nagiza F. Samatova</i>	
Fast Multiresolution Reads of Massive Simulation Datasets	314
<i>Sidharth Kumar, Cameron Christensen, John A. Schmidt, Peer-Timo Bremer, Eric Brugger, Venkatram Vishwanath, Philip Carns, Hemanth Kolla, Ray Grout, Jacqueline Chen, Martin Berzins, Giorgio Scorzelli, and Valerio Pascucci</i>	

Rebasing I/O for Scientific Computing: Leveraging Storage Class Memory in an IBM BlueGene/Q Supercomputer	331
<i>Felix Schürmann, Fabien Delalondre, Pramod S. Kumbhar, John Biddiscombe, Miguel Gila, Davide Tacchella, Alessandro Curioni, Bernard Metzler, Peter Morjan, Joachim Fenkes, Michele M. Franceschini, Robert S. Germain, Lars Schneidenbach, T.J. Christopher Ward, and Blake G. Fitch</i>	
Orthrus: A Framework for Implementing Efficient Collective I/O in Multi-core Clusters	348
<i>Xuechen Zhang, Jianqiang Ou, Kei Davis, and Song Jiang</i>	
Fast and Energy-efficient Breadth-First Search on a Single NUMA System	365
<i>Yuichiro Yasui, Katsuki Fujisawa, and Yukinori Sato</i>	
Evaluation of the Impact of Direct Warm-Water Cooling of the HPC Servers on the Data Center Ecosystem	382
<i>Radosław Januszewski, Norbert Meyer, and Joanna Nowicka</i>	
A Case Study of Energy Aware Scheduling on SuperMUC	394
<i>Axel Auweter, Arndt Bode, Matthias Brehm, Luigi Brochard, Nicolay Hammer, Herbert Huber, Raj Panda, Francois Thomas, and Torsten Wilde</i>	
Exploiting SIMD and Thread-Level Parallelism in Multiblock CFD	410
<i>Ioan Hadade and Luca di Mare</i>	
The Performance Characterization of the RSC PetaStream Module	420
<i>Andrey Semin, Egor Druzhinin, Vladimir Mironov, Alexey Shmelev, and Alexander Moskovsky</i>	
Deploying Darter - A Cray XC30 System	430
<i>Mark R. Fahey, Reuben Budiardja, Lonnie Crosby, and Stephen McNally</i>	
Cyme: A Library Maximizing SIMD Computation on User-Defined Containers	440
<i>Timothée Ewart, Fabien Delalondre, and Felix Schürmann</i>	
A Compiler-Assisted OpenMP Migration Method Based on Automatic Parallelizing Information	450
<i>Kazuhiko Komatsu, Ryusuke Egawa, Hiroyuki Takizawa, and Hiroaki Kobayashi</i>	
A Type-Oriented Graph500 Benchmark	460
<i>Nick Brown</i>	

A Dynamic Execution Model Applied to Distributed Collision Detection	470
<i>Matthew Anderson, Maciej Brodowicz, Luke Dalessandro, Jackson DeBuhr, and Thomas Sterling</i>	
Implementation and Optimization of Three-Dimensional UPML-FDTD Algorithm on GPU Clusters	478
<i>Lei Xu and Ying Xu</i>	
Real-Time Olivary Neuron Simulations on Dataflow Computing Machines	487
<i>Georgios Smaragdos, Craig Davies, Christos Strydis, Ioannis Sourdis, Cătălin Ciobanu, Oskar Mencer, and Chris I. De Zeeuw</i>	
Tofu Interconnect 2: System-on-Chip Integration of High-Performance Interconnect	498
<i>Yuichiro Ajima, Tomohiro Inoue, Shinya Hiramoto, Shunji Uno, Shinji Sumimoto, Kenichi Miura, Naoyuki Shida, Takahiro Kawashima, Takayuki Okamoto, Osamu Moriyama, Yoshiro Ikeda, Takekazu Tabata, Takahide Yoshikawa, Ken Seki, and Toshiyuki Shimizu</i>	

Poster Extended Abstracts

Compression by Default – Reducing Total Cost of Ownership of Storage Systems	508
<i>Michael Kuhn, Konstantinos Chasapis, Manuel F. Dolz, and Thomas Ludwig</i>	
Predictive Performance Tuning of OpenACC Accelerated Applications	511
<i>Shahzeb Siddiqui and Saber Feki</i>	
Particle-in-Cell Plasma Simulation on CPUs, GPUs and Xeon Phi Coprocessors	513
<i>Sergey Bastrakov, Iosif Meyerov, Igor Surmin, Evgeny Efimenko, Arkady Gonoskov, Alexander Malyshев, and Mikhail Shiryaev</i>	
Application Tracking Using the Ichnaea Tools	515
<i>Iain Miller, Andy Bennett, and Oliver Perks</i>	
OpenFFT: An Open-Source Package for 3-D FFTs with Minimal Volume of Communication	517
<i>Truong Vinh Truong Duy and Taisuke Ozaki</i>	
Author Index	519

Sustained Petascale Performance of Seismic Simulations with SeisSol on SuperMUC

Alexander Breuer¹, Alexander Heinecke¹, Sebastian Rettenberger¹,
Michael Bader¹, Alice-Agnes Gabriel², and Christian Pelties²

¹ Department of Informatics, Technische Universität München, Germany

² Department of Earth and Environmental Sciences, Geophysics,
Ludwig-Maximilians-Universität München, Germany

Abstract. Seismic simulations in realistic 3D Earth models require peta- or even exascale computing power to capture small-scale features of high relevance for scientific and industrial applications. In this paper, we present optimizations of SeisSol – a seismic wave propagation solver based on the Arbitrary high-order accurate DERivative DiScontinuous Galerkin (ADER-DG) method on fully adaptive, unstructured tetrahedral meshes – to run simulations under production conditions at petascale performance. Improvements cover the entire simulation chain: from an enhanced ADER time integration via highly scalable routines for mesh input up to hardware-aware optimization of the innermost sparse-/dense-matrix kernels. Strong and weak scaling studies on the SuperMUC machine demonstrated up to 90% parallel efficiency and 45% floating point peak efficiency on 147k cores. For a simulation under production conditions (10^8 grid cells, $5 \cdot 10^{10}$ degrees of freedom, 5 seconds simulated time), we achieved a sustained performance of 1.09 PFLOPS.

Keywords: seismic wave and earthquake simulations, petascale, vectorization, ADER-DG, parallel I/O.

1 Introduction and Related Work

The accurate numerical simulation of seismic wave propagation through a realistic three-dimensional Earth model is key to a better understanding of the Earth's interior. It is thus utilized extensively for earthquake simulation and seismic hazard estimation and also for oil and gas exploration [15, 34, 47]. Although the physical process of wave propagation is well understood and can be described by a system of hyperbolic partial differential equations (PDEs), the numerical simulation of realistic settings still poses many methodological and computational challenges. These may include complicated geological material interfaces, faults, and topography. In particular, to resolve the high-frequency content of the wave field, which is desired for capturing crucial small-scale features, computational resources on peta- and probably exascale level are required [16, 28, 45].

Therefore, respective simulation software must be based on geometrically flexible and high-order accurate numerical methods combined with highly scalable and computationally efficient parallel implementations.

In this paper we present optimizations of the seismic wave propagation software SeisSol to realize such high-order adaptive simulations at petascale performance. SeisSol is based on the Arbitrary high-order accurate DERivative Discontinuous Galerkin (ADER-DG) method on unstructured tetrahedral meshes [8]. It has been successfully applied in various fields of seismology [23, 41, 50], exploration industry [24] and earthquake physics [36, 37, 39] – demonstrating advantages whenever the simulations need to account for highly complicated geometrical structures, such as topography or the shapes of geological fault zones.

With our optimizations, we consequently target high-frequency, large-scale seismic forward simulations. To achieve high sustained performance on respective meshes consisting of 10^8 – 10^9 elements with resulting small time step sizes, it is essential to leverage all levels of parallelism on the underlying supercomputer to their maximum extent. The related question of sustained peak performance on application level has received growing interest over the last years [26, 33]. For example, the BlueWaters project aims on sustained petascale performance on application level and thus chose to not have their supercomputer listed in the Top500 list [31], despite its aggregate peak performance of 13 PFLOPS.

The demand for sustained performance at the petascale and scaling to hundreds of thousands of cores is also well reflected by related work on large-scale seismic simulations. Cui et al. [5] demonstrated respective earthquake and seismic hazard simulations, recently also utilizing heterogeneous platforms: for simulations on the GPU-accelerated Titan supercomputer, they reported a speed-up of 5 due to using GPUs [4]. That the reported performance is less than 5% of the theoretical peak performance of the GPU accelerators shows the challenge of optimizing production codes for these platforms. SpecFEM [40], a well-known community seismic simulation code, has been finalist in the Gordon Bell Award 2008 for simulations on 62k cores [3] and an achieved 12% peak performance on the Jaguar system at Oak Ridge National Laboratory. In 2013, SpecFEM was executed with 1 PFLOPS sustained performance on the BlueWaters supercomputer utilizing 693,600 processes on 21,675 compute nodes which provide 6.4 PFLOPS theoretical peak performance¹.

To achieve sustained petascale performance with SeisSol, optimizations in its entire simulation chain were necessary. In Sec. 2, we outline the ADER-DG solver for the wave equations and present an improved ADER time integration scheme that substantially reduced the time to solution for this component. Sec. 3 describes a highly-scalable mesh reader and the hardware-aware optimization of innermost (sparse and dense) matrix operators – two key steps for realizing simulations with more than a billion grid cells at a performance of 1.4 PFLOPS (44% peak efficiency) on the SuperMUC machine. Respective strong and weak scaling tests, as well as a high-resolution benchmark scenario of wave propagation

¹ <http://www.ncsa.illinois.edu/news/stories/PFapps/>

in the Mount Merapi volcano under production conditions, are presented in Sec. 4.

2 An ADER-DG Solver for the Elastic Wave Equation

SeisSol solves the three-dimensional elastic wave equations, a system of hyperbolic PDEs, in velocity-stress formulation. In the following we give a short description of the fully-discrete update scheme, with focus on issues relevant for implementation at high performance. Details about the mathematical derivation of the underlying equations, handling of boundary conditions, source terms, viscoelastic attenuations, and more of SeisSol's features, can be found in [7–9, 25].

For spatial discretization SeisSol employs a high-order Discontinuous Galerkin (DG) Finite Element method with the same set of hierarchical orthogonal polynomials as ansatz and test functions [20, 25] on flexible unstructured tetrahedral meshes. For every tetrahedron T_k , the matrix $Q_k(t)$ contains the degrees of freedom (DOFs) that are the time-dependent coefficients of the modal basis on the unique reference tetrahedron for all nine physical quantities (six stress and three velocity components) [8]. The ADER time integration in SeisSol allows arbitrary high order time discretization. Adopting the same convergence order \mathcal{O} in space and time results in convergence order \mathcal{O} for the overall ADER-DG scheme. $\mathcal{O} = 5$ and $\mathcal{O} = 6$ are the typically chosen orders for production runs.

2.1 Compute Kernels and Discrete Update Scheme

The ADER-DG scheme in SeisSol is formulated as time, volume and boundary integrations applied to the element-local matrices $Q_k^n = Q_k(t^n)$, which contain the DOFs for tetrahedron T_k at time step t^n . Application of the different integrations leads to the DOFs Q_k^{n+1} at the next time step t^{n+1} . In fully explicit formulation each of these integration steps is formulated as a compute kernel that may be expressed as a series of matrix-matrix multiplications.

Time Kernel: The first compute kernel, consisting of the ADER time integration, derives an estimate $\mathcal{I}(t^n, t^{n+1}, Q_k^n)$ of the DOFs Q_k^n integrated in time over the interval $[t^n, t^{n+1}]$:

$$\mathcal{I}_k^{n,n+1} := \mathcal{I}_k(t^n, t^{n+1}, Q_k^n) = \sum_{j=0}^{\mathcal{O}-1} \frac{(t^{n+1} - t^n)^{j+1}}{(j+1)!} \frac{\partial^j}{\partial t^j} Q_k(t^n), \quad (1)$$

where the time derivates $\partial^j / \partial t^j Q_k(t^n)$ are computed recursively by:

$$\frac{\partial^{j+1}}{\partial t^{j+1}} Q_k = -\hat{K}^\xi \left(\frac{\partial^j}{\partial t^j} Q_k \right) A_k^* - \hat{K}^\eta \left(\frac{\partial^j}{\partial t^j} Q_k \right) B_k^* - \hat{K}^\zeta \left(\frac{\partial^j}{\partial t^j} Q_k \right) C_k^*, \quad (2)$$

with initial condition $\partial^0 / \partial t^0 Q_k(t^n) = Q_k^n$. Matrices $\hat{K}^{\xi_c} = M^{-1} (K^{\xi_c})^T$, with the reference coordinates $\xi_1 = \xi$, $\xi_2 = \eta$ and $\xi_3 = \zeta$, are the transpose of the stiffness matrices K^{ξ_c} over the reference tetrahedron multiplied (during preprocessing) by the inverse diagonal mass matrix M^{-1} . A_k^* , B_k^* and C_k^* are linear combinations of the element-local Jacobians.

Volume Kernel: SeisSol's volume kernel $\mathcal{V}_k(\mathcal{I}_k^{n,n+1})$ accounts for the local propagation of the physical quantities inside each tetrahedron using the time integrated unknowns $\mathcal{I}(t^n, t^{n+1}, Q_k^n)$ computed by the time kernel expressed in Eqs. (1) and (2):

$$\mathcal{V}_k\left(\mathcal{I}_k^{n,n+1}\right) = \tilde{K}^\xi\left(\mathcal{I}_k^{n,n+1}\right) A_k^* + \tilde{K}^\eta\left(\mathcal{I}_k^{n,n+1}\right) B_k^* + \tilde{K}^\zeta\left(\mathcal{I}_k^{n,n+1}\right) C_k^*. \quad (3)$$

Analog to the time kernel, the matrices $\tilde{K}^{\xi_c} = M^{-1}K^{\xi_c}$ are defined over the reference tetrahedron. Matrices A_k^* , B_k^* and C_k^* are defined as in Eq. (2).

Boundary Kernel: The boundary integration connects the system of equations inside each tetrahedron to its face-neighbors. Here the DG-characteristic Riemann problem is solved, which accounts for the discontinuity between the element's own quantities and the quantities of its face-neighbors. The kernel uses the time integrated DOFs $\mathcal{I}_k^{n,n+1}$ of the tetrahedron T_k itself and $\mathcal{I}_{k(i)}^{n,n+1}$ of the four face-neighbors $T_{k(i)}$, $i = 1, \dots, 4$:

$$\begin{aligned} \mathcal{B}_k\left(\mathcal{I}_k^{n,n+1}, \mathcal{I}_{k(1)}^{n,n+1}, \dots, \mathcal{I}_{k(4)}^{n,n+1}\right) &= \sum_{i=1}^4 \left(M^{-1}F^{-,i}\right) \mathcal{I}_k^{n,n+1}\left(\frac{|S_k|}{|J_k|} N_{k,i} A_k^+ N_{k,i}^{-1}\right) \\ &\quad + \sum_{i=1}^4 \left(M^{-1}F^{+,i,j_k(i),h_k(i)}\right) \mathcal{I}_{k(i)}^{n,n+1}\left(\frac{|S_k|}{|J_k|} N_{k,i} A_{k(i)}^- N_{k,i}^{-1}\right). \end{aligned} \quad (4)$$

$F^{-,i}$ and $F^{+,i,j_k(i),h_k(i)}$ are 52 integration matrices defined over the faces of the reference tetrahedron [8]. The choice of these matrices depends on the three indices $i = 1 \dots 4$, $j_k(i) = 1 \dots 4$ and $h_k(i) = 1 \dots 3$, which express the different orientations of two neighboring tetrahedrons relative to each other with respect to their projections to the reference tetrahedron [1]. $N_{k,i} A_k^+ N_{k,i}^{-1}$ and $N_{k,i} A_{k(i)}^- N_{k,i}^{-1}$ denote the element-local flux solvers, multiplied by the scalars $|J_k|$ and $|S_k|$, which are the determinant of the Jacobian of the transformation to reference space $\xi - \eta - \zeta$ and the area of the corresponding face, during initialization. The flux solvers solve the face-local Riemann problems by rotating the time integrated quantities to the face-local spaces and back via the transformation matrices $N_{k,i}$ and $N_{k,i}^{-1}$.

Update Scheme: Combining time, volume and boundary kernel, we get the complete update scheme from one time step level t^n to the next t^{n+1} :

$$Q_k^{n+1} = Q_k^n - \mathcal{B}_k\left(\mathcal{I}_k^{n,n+1}, \mathcal{I}_{k(1)}^{n,n+1}, \dots, \mathcal{I}_{k(4)}^{n,n+1}\right) + \mathcal{V}_k\left(\mathcal{I}_k^{n,n+1}\right) \quad (5)$$

Matrix Computations: SeisSol's update scheme is heavily dominated by matrix-matrix operations. The size $B_{\mathcal{O}} \times B_{\mathcal{O}}$ of all matrices over the reference tetrahedron – \hat{K}^{ξ_c} , \tilde{K}^{ξ_c} , $F^{-,i}$ and $F^{+,i,j_k(i),h_k(i)}$ – depends on the order \mathcal{O} of the scheme. In turn, the order \mathcal{O} defines the number of used basis functions $B_{\mathcal{O}}$: $B_1 = 1$, $B_2 = 4$, $B_3 = 10$, $B_4 = 20$, $B_5 = 35$, $B_6 = 56$, $B_7 = 84$, The

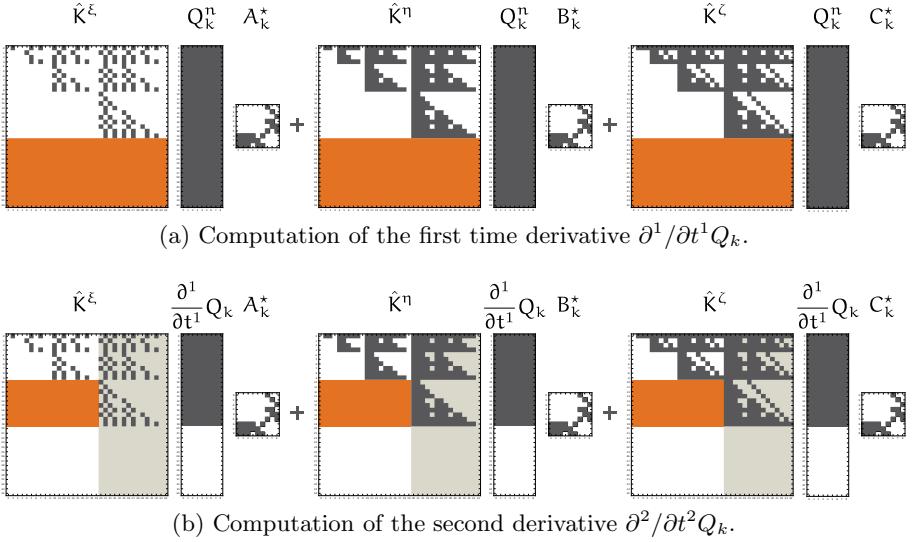


Fig. 1. First two recursions of the ADER time integration for a fifth-order method. Orange blocks generate zero blocks in the derivatives, light-gray blocks hit zero blocks.

element-local matrices – A^* , B^* , C^* , $N_{k,i}A_k^+N_{k,i}^{-1}$ and $N_{k,i}A_{k(i)}^-N_{k,i}^{-1}$ – are of size 9×9 , which correlates to the nine quantities of the elastic wave equations. A detailed description of the involved sparsity patterns is given in [1].

2.2 Efficient Evaluation of the ADER Time Integration

In this subsection, we introduce an improved scheme that reduces the computational effort of the ADER time integration formulated in [25] and in discrete form in Sec. 2.1.

Analyzing the sparsity patterns of the involved stiffness matrices, we can substantially reduce the number of operations in the ADER time integration. This is especially true for high orders in space and time: The transposed stiffness matrices $(K^\xi)^T$, $(K^\eta)^T$ and $(K^\zeta)^T$ of order \mathcal{O} contain a zero block starting at the first row that accounts for the new hierarchical basis functions added to those of the preceding order $\mathcal{O}-1$. According to Eq. (1) we have to compute the temporal derivatives $\partial^j/\partial t^j Q_k$ for $j \in 1 \dots \mathcal{O}-1$ by applying Eq. (2) recursively to reach approximation order \mathcal{O} in time.

The first step (illustrated in Fig. 1a) computes $\partial^1/\partial t^1 Q_k$ from the initial DOFs Q_k^n . The zero blocks of the three matrices \hat{K}^{ξ_c} generate a zero block in the resulting derivative and only the upper block of size $B_{\mathcal{O}-1} \times 9$ contains non-zeros. In the computation of the second derivative $\partial^2/\partial t^2 Q_k$ we only have to account for the top-left block of size $B_{\mathcal{O}-1} \times B_{\mathcal{O}-1}$ in the matrices \hat{K}^{ξ_c} . As illustrated in Fig. 1b the additional non-zeros hit the previously generated zero block of derivative $\partial^1/\partial t^1 Q_k$. The following derivatives $\partial^j/\partial t^j Q_k$ for $j \in 3 \dots \mathcal{O}-1$

proceed analogously and for each derivative j only the \hat{K}^{ξ_c} -sub-blocks of size $B_{O-j+1} \times B_{O-j+1}$ have to be taken into account. Obviously, the zero blocks also appear in the multiplication with the matrices A^* , B^* and C^* from the right and additional operations can be saved.

3 Optimizing SeisSol for Sustained Petascale Performance

In order to ensure shortest execution time and highest efficiency of SeisSol, all levels of parallelism of the considered petascale machine have to be leveraged. This starts with an efficient usage of the parallel file system and ends at a carefully tuned utilization of each core’s vector instruction set. Additionally, we have to ensure that the problem’s decomposition is efficiently mapped onto the computing resources. The state-of-the-art is to join several thousands of shared memory multi-core systems to large distributed-memory cluster installations. This system layout needs to be addressed for best performance, cf. [42, 43]: the shared memory within each node must be leveraged, which results in a faster intra-node communication and less partitions of the problem under investigation. Friedley et al. [10] presented strategies on how to address the first item within a pure distributed-memory programming model (MPI) [30]. Although such an approach utilizes the fast shared memory for intra-node communications, it still requires a partitioning of the problem handling all processing elements separately. We address this issue in SeisSol by switching to the explicit shared-memory programming model OpenMP [35] for parallel subtasks, such as loops over all process-local elements or gathering elements from the unstructured mesh into data exchange buffers.

In the following two subsections, we present the two major steps of our rigorous performance re-engineering process of turning SeisSol into a sustained petascale application, even on entry-level petascale systems. First, we describe how to read input data in a highly-parallel and efficient manner, and second, we elaborate how to achieve close-to-peak node-level performance by a highly-tuned BLAS3 implementation for dense and sparse matrix operations of small rank.

3.1 Highly Scalable Mesh Reader

SeisSol’s approach is especially well-suited for the simulation of seismic wave propagation in highly complex geometries and heterogeneous 3D Earth models, relying on the flexibility offered by fully adaptive, unstructured tetrahedral meshes [8]. Comparable approaches, which first generate coarse unstructured meshes and successively refine these in a structured way (such as [2, 13]), do not provide the same approximation quality in terms of geometry and thus in the accuracy of the obtained results [38].

SeisSol’s workflow for the generation and parallel input of high-resolution discretization grids requires three main steps: (1) Starting from a sufficiently detailed CAD model of the system geometry, a fully adaptive, unstructured tetrahedral mesh needs to be generated using a suitable meshing software; the desired

high-quality meshes (e.g., without degenerated mesh cells) consisting of 10s to 100s of million mesh cells can only be generated by few packages – our preferred software being SimModeler by Simmetrix². The mesh information is stored to be used for multiple simulations: for these mesh files we used the ASCII-based GAMBIT format, so far. (2) To generate compact and balanced parallel partitions, we use the multilevel k -way partitioning algorithm [21, 22] implemented in ParMETIS (or similar partitioning software). The output is typically a partitioning file that maps each grid cell to a specific partition. (3) From the mesh file and the partition-mapping file, not only the partitions need to be generated for each MPI process, but also the communication structures and ghost layers (virtual elements of neighboring MPI domains) for communication. In previous versions of SeisSol, this step had been part of the initialization phase. However, it limited simulations to meshes with only a few million grid cells, because computing the required information did not scale to several thousand MPI ranks and did not allow parallel I/O.

In order to retain the flexibility regarding mesh generator and partitioning software for individual users, we decided not to integrate the mesh generator into SeisSol, as for example proposed in [3]. Instead, we reorganized steps (2) and (3) of our workflow into an offline part to compute partitions and communication structures, and a highly-scalable input phase that remains in SeisSol.

In the offline part, we read the GAMBIT mesh file and use ParMETIS to construct the so-called dual graph of the mesh. As the dual graph reflects the data exchange between grid cells in the ADER-DG scheme, both the partitioning and the computation of communication structures are based on this graph. After the partitioning with ParMETIS, we sort the elements and vertices according to their partition and duplicate vertices that belong to multiple partitions. The small overhead due to boundary vertex doubling easily pays off by the simplifications in the resulting new format. Then, we precompute the communication structures and ghost layers required by SeisSol from the dual graph. The ordered elements and vertices are stored together with the precomputed data in a customized format. This new file format is based on netCDF [44], a generic binary file format for multidimensional arrays that supports parallel I/O. Our offline tool is parallelized to satisfy the memory requirements of large meshes. To convert a mesh with 99,831,401 tetrahedral elements for our strong scaling setup (Sec. 4.2) from GAMBIT to netCDF we required 47.8 minutes on 64 cores and consumed 32.84 GB memory. 65% of this time was spent reading the original file.

Reading the netCDF file during the redesigned input phase exploits netCDF’s efficient MPI-I/O-based implementation and strongly profits from parallel file systems available on the respective supercomputer. For our largest simulation with a mesh of 8,847,360,000 tetrahedrons the parallel mesh input required only 23 seconds on 9,216 nodes (using the GPFS file system, see the description of the weak scaling setup, Sec. 4.1).

² <http://simmetrix.com/>

3.2 (Sparse-)DGEMM Functions for Matrices of Small Rank

According to the equations and matrix structures elaborated in Sec. 2, several matrix operations with different sparsity patterns and small ranks had to be optimized. Before the re-engineering process, SeisSol stored all sparse matrices in a simple coordinate format, regardless of the actual sparsity pattern of a certain matrix. Dense matrix structures were only used for the DOFs, Q_k . The goal of our optimization was to replace the kernel operations by high-performance and specialized (sparse-)DGEMM functions within SeisSol. Throughout the integration schemes (compare Sec. 2.1), the following types of matrix-matrix multiplications are required: $\text{sparse} \times \text{dense} = \text{dense}$, $\text{dense} \times \text{sparse} = \text{dense}$ and $\text{dense} \times \text{dense} = \text{dense}$.

In previous work [1], we adopted the original approach of SeisSol and handled all matrices except DOF matrices as sparse. Since the sparsity patterns of all matrices are known up-front, we generated specialized sparse-DGEMM routines that efficiently leverage the hardware’s vector instruction extensions. This was achieved by hard-wiring the sparsity pattern of each operation into the generated code through unrolling [29]. Note that such a generation has to take place in an offline step, because compilers’ auto-vectorizer, source-to-source vectorizers such as Scout [27] or highly-efficient BLAS libraries such as Intel MKL [19] or Blaze [18] regard the index-structures of the sparse matrices as variables since they are runtime parameters. Through this approach we were able to accelerate SeisSol by a factor of 2.5–3.0 as presented in [1].

Besides handling either operand matrix A or B of a DGEMM call as sparse, it is profitable to generate optimal code for dense matrix kernels as well, because many of the operands feature dense blocks, cf. [32, 49, 53]. Due to the small size of the matrices (56×56 , 56×9 , 9×9 for $\mathcal{O} = 6$), calling highly-efficient BLAS3 functions offered by vendor implementations does not lead to satisfying results. From Sec. 2 we know that the number of columns N of at least two operands is always 9, so we hard-wired $N = 9$ into our code. Additionally, this optimization prevents reusing inner-most block-operations of processing 4×4 blocks or 8×4 blocks as proposed in [14, 54, 55]. However, we applied identical ideas to an inner-most kernel of 12×3 blocks which perfectly match with the size of Intel’s AVX registerfile (16 32-byte registers).

In order to select the fastest alternative between the operation-specific sparse kernel and its highly-tuned dense counterpart, we extended our time, volume and boundary kernels by a sparse-dense switch: during initialization we hard-wire function pointers to an optimal matrix kernel (sparse or dense) for each individual matrix operator appearing in the different kernels. In order to decide whether sparse or dense kernels for a specific matrix lead to shorter time to solution, we performed dry-runs on the first 100 time steps of the representative SCEC LOH.1 benchmark [6] with 7,252,482 elements for approximation orders $\mathcal{O} \in \{2, 3, \dots, 6\}$. Such a tuning process for linear algebra is well-known from projects such as ATLAS [51, 52] or OSKI [48]. For each run we identified the percentage of non-zeros required to make the corresponding matrix kernel perform better as dense (instead of sparse) kernel. These automated training

runs in the preparation phase of SeisSol returned the desired collection of kernel operations that provides the shortest time to solution. The identified kernels were used afterwards to compile the final SeisSol binary for a certain order of approximation. A detailed discussion of this switch can be found in [17], which in general suggests to switch to the discussed sparse computing kernels if more than 80% of the matrix's entries are zeros.

In conclusion we want to highlight that both of the implemented kernel variants support the improved ADER time integration scheme presented in Sec. 2.2. This was realized by injecting jumps that skip the processing of unused sub-blocks in the case of generated sparse-DGEMM kernels. When using their dense counterpart, no change was required as our highly-optimized DGEMM kernel features variable values for M and K (BLAS notation). However, M is rounded towards the next multiple of the blocking width of our inner-most kernel. Such a ‘zero-padding’ is required to ensure best-possible performance, as it reduces the complexity of the instruction mix and thus optimally exploits level 1 instruction caches. Note that such a rounding is not needed along K since there is no vectorization in place.

4 Performance Evaluation on SuperMUC

In this section we analyze the behavior of SeisSol in strong and weak scaling benchmarks executed on SuperMUC³ at Leibniz Supercomputing Centre. SuperMUC features 147,456 cores and is at present one of the biggest Intel Sandy Bridge systems worldwide. It comprises two eight-core Intel Xeon E5-2680 processors per node at 2.7GHz. With a collective theoretical double-precision peak performance of more than 3 PFLOPS, it was ranked #10 on the November 2013 Top500 list [31]. In contrast to supercomputers offered by Cray, SGI or IBM’s BlueGene, the machine is based on a high-performance Infiniband commodity network organized in a fat tree with 18 islands that consist of 512 nodes each. All nodes can communicate within each island at full IB-FDR-10 data-rate. In case of inter-island communication, four nodes share one uplink to the spine switch, leading to reduced bandwidth for inter-island communication.

For all of the following results we derived the number of double-precision floating point operations (FLOP) in a preprocessing step. In addition, we validated the obtained numbers by aggregating FLOP-counters integrated into SeisSol’s compute kernels. We followed this combined approach for all strong-scaling runs and small weak-scaling runs (< 512 cores). For reasons of practicability we restricted the application of the runtime method to smaller weak-scaling jobs. The obtained floating point operations per second (FLOPS) rates allow us to calculate directly the peak efficiency of SeisSol on SuperMUC.

Before scaling SeisSol to the full SuperMUC petascale machine, we again ran the SCEC LOH.1 benchmark in a strong-scaling setting to determine the performance increase of our optimized version of SeisSol in comparison to the classical SeisSol implementation in Fig. 2. We see a factor-5 improvement in time to

³ <http://www.lrz.de/services/compute/supermuc/>

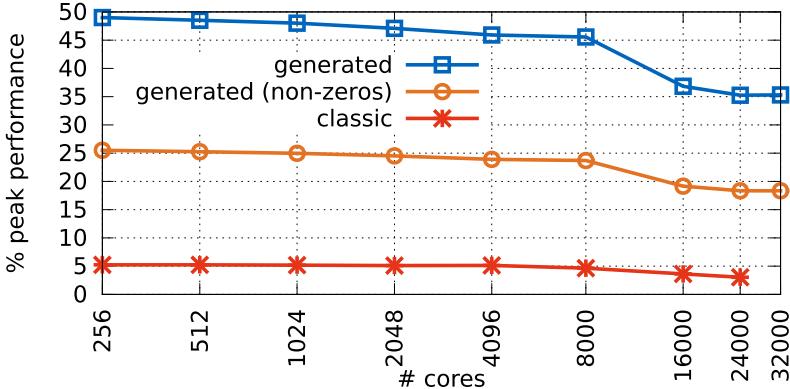


Fig. 2. Strong scaling of the SCEC LOH.1 benchmark with 7,252,482 elements using approximation order $\mathcal{O} = 6$ for the classic version of SeisSol (red) and our highly tuned derivate (blue and orange). The slight efficiency decrease for $>8,000$ cores is mostly due to SuperMUC’s island concept.

solution, which translates into 25% peak performance without accounting for vector instruction set padding and nearly 50% machine efficiency on vector instruction level. Note that the kink in all three performance series, when increasing the core count from 8,000 to 16,000 cores, is due to SuperMUC’s island concept. Besides a raw performance gain of $5\times$ on the same numbers of cores, our optimized version of SeisSol is able to strong-scale to larger numbers of cores due to its hybrid OpenMP and MPI parallelization approach leading to an aggregate speedup of nearly 10 in this setting.

4.1 Weak Scaling

For a weak scaling study, and also to test SeisSol’s limits in terms of mesh size, we discretized a cubic domain with regularly refined tetrahedral meshes. The computational load was kept constant with 60,000 elements per core and a total of 100 time steps per simulation. To complete the setup, we used a sinusoidal wave as initial condition and 125 additional receivers distributed across the domain. We performed the weak scaling simulations starting at 16 cores as a baseline. Then, we doubled the number of cores in every next run up to 65,536 cores and performed an additional run utilizing all of the available 147,456 cores. Fig. 3 shows the fraction of the theoretical peak performance. We observed the maximum performance using all 147,456 cores with a sustained performance of 1.42 PFLOPS, which correlates to 44.5% of theoretical peak performance and a parallel efficiency of 89.7%.

In contrast to the observations made for the LOH.1 strong scaling benchmark presented in the previous section, a performance kink is identifiable when more than two islands are employed, specifically, when moving from 16,000 to

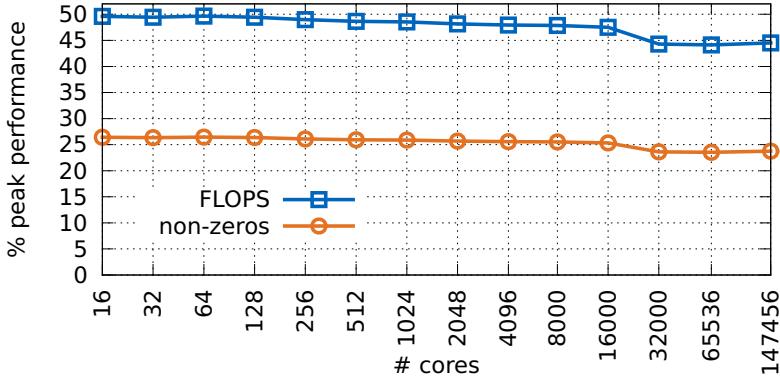


Fig. 3. Weak scaling of the cube benchmark with 60,000 elements per core using approximation order $\mathcal{O} = 6$. Shown are hardware FLOPS (blue) and non-zero operations (orange). The slight efficiency decrease for $>16,000$ cores is mostly due to SuperMUC's island concept.

32,000 cores. Since inter-island communication only happens in one dimension of the computational domain and we performed a weak scaling study, we have lower requirements on the communication infrastructure compared to the LOH.1 benchmark. Therefore, in the case of two islands only 100 ($< \frac{1}{4}$ of the nodes per island) inter-island communication channels are used. However, when utilizing four or more islands the 128 up-links per island, given by the islands' pruned tree organization, become a limiting factor as these runs demand more than 200 channels.

4.2 Strong Scaling

The strong scaling setup was based on a typical scenario setup as used in geophysical forward modeling. We discretized the volcano Mount Merapi (Java, Indonesia) with a total number of 99,831,401 tetrahedrons. The setup contained two layers of different material properties (density, seismic wave speeds), topography obtained from local digital elevation models [11, 12] and a moment tensor representation of a double-couple seismic point source approximation. We elaborate the geophysical specifications in Sec. 4.3, where a run under production conditions is presented.

In the strong scaling runs, we performed 1000 time steps on 1024 to 65,536 cores by doubling the number of cores in every next run. Finally, an additional run on all 147,456 cores was executed. In the extreme case of using the entire SuperMUC, the load per core was only ≈ 677 elements. Fig. 4 shows the fraction of theoretical peak performance for all runs. The sustained performance on all 147,456 cores was 1.13 PFLOPS, which is 35.58% of theoretical peak performance and correlates to 19.42%, if only non-zero operations are considered. The parallel efficiency with 1024 cores as baseline was at 73.04%.

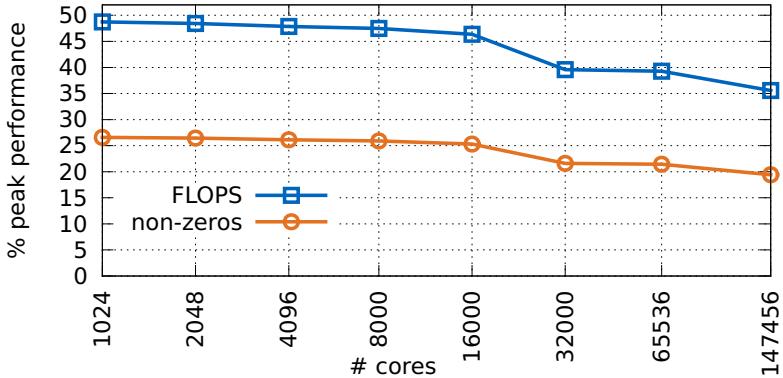


Fig. 4. Strong scaling of the Mount Merapi benchmark with 99,831,401 tetrahedrons using approximation order $\mathcal{O} = 6$. Shown are hardware FLOPS (blue) and non-zero operations (orange).

4.3 Production Run

Finally, we performed a simulation under production conditions on all 147,456 cores of SuperMUC. As a test case scenario, we chose the volcano Mount Merapi to demonstrate the solver's capabilities and that its optimizations regarding parallel I/O, hybrid parallelization as well as vectorization are sustained for arbitrary mesh and model complexity.

Several eruptions of Mount Merapi have caused fatalities and the stratovolcano is still highly active. Thus, Mount Merapi is subject of ongoing research and a network of eight seismographs monitors tremors and earthquakes, with the goal of implementing a functional early warning system in the future. Seismic forward modeling could help to locate such tremors and to gain a better image of the geological subsurface structure. Therefore, synthetic time series and 3D wave field visualization are required to support the interpretation.

Our simulation setup was identical to the strong scaling runs, with the addition of a set of 59 receivers distributed throughout the domain, sampling the physical wave field every 0.001 seconds. Also, in contrast to the strong scaling runs, we now computed 166,666 time steps to accomplish 5 seconds in simulation time. The total simulation took 3 hours and 7.5 minutes with 1 minute and 22 seconds being initialization. The remainder was spent in the time marching loop, which ran at 1.09 PFLOPS. This correlates to 34.14% theoretical peak performance. Thus, we conclude that our optimization can be fully exploited under realistic research conditions.

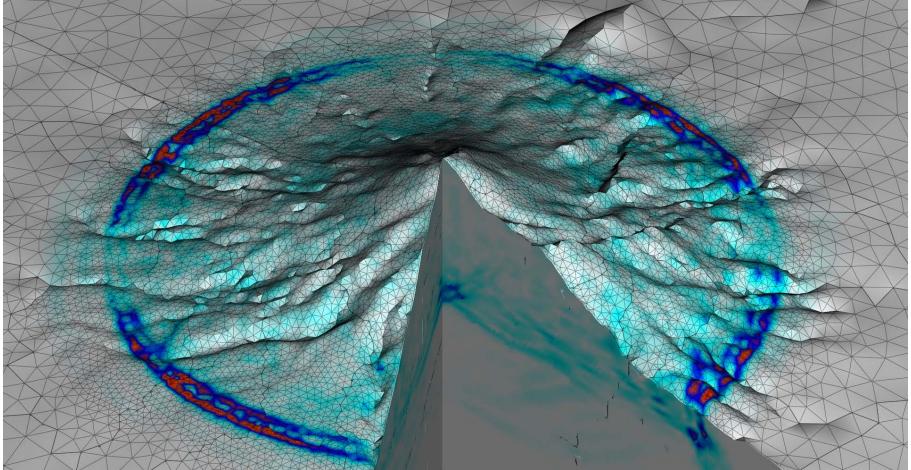


Fig. 5. Wave field of the Mount Merapi benchmark after 4 seconds. For illustration purposes a smaller mesh with 1,548,496 tetrahedrons was used. The tetrahedral approximation of the topography is shown as overlay on the surface. Insight into the interior is provided by virtually removing a section in the front. Warmer colors denote higher wave field energy.

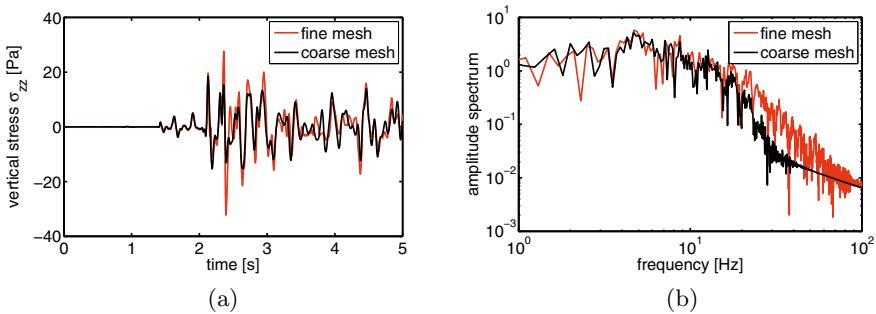


Fig. 6. In (a) we compare the time series produced with the fine mesh (red) and the coarse mesh (black). Clearly, the fine mesh shows larger amplitudes (e.g. at second 2.3) that might be damped out for the coarse mesh simulation due to increased numerical dissipation. More difficult to identify, but visible in the spectrum (b), is the increased frequency content beyond 20 Hz of the fine mesh.

Fig. 5 illustrates the model setup and shows the wave field at 4 seconds simulation time. The achieved high resolution (due to applied order and element size) allows accurate modeling of the highest frequencies (more than 20 Hz) capturing small-scale details in the wave-form modulation due to topography or wave interaction at material contrasts. Wave field complexity is already visible in Fig. 5, although the image is based on data from a much coarser mesh for

visualization purposes. In Fig. 6 we compare results from the presented high-resolution simulation with a coarser simulation. The coarser mesh discretized the model domain by 100 m in the shallower region and 500 m in the remainder resulting in 1,548,496 tetrahedrons, whereas the fine meshes used an element edge length of 28 m and 100 m, respectively resulting in 99,831,401 tetrahedrons. Both simulations were performed with $\mathcal{O} = 6$. The receiver station for Fig. 6 was randomly picked and is located in the interior of the volcano. As an example, we plot the vertical stress component σ_{zz} . In (a) the time series are plotted, showing a higher amount of wave form details for the fine-mesh simulation. The increased frequency content is also reflected by plotting the spectrum (b) of the time series. Due to the decreased numerical errors by using the fine mesh more details and small scale features are uncovered.

5 Conclusions

Performing a head-to-toe performance re-engineering, we have enabled SeisSol for seismic simulations at petascale. The improved mesh input allows SeisSol to run simulation settings on meshes with billions of grid cells and (for order $\mathcal{O} = 6$) more than 10^{12} degrees of freedom. Based on the generation of optimized sparse- and dense-matrix kernels, we achieved 20–25% effective peak efficiency, i.e., disregarding instructions due to vector padding or dense computations. These results demonstrate that SeisSol is competitive with earlier discussed codes reaching 10–16% peak efficiency. The aggregate performance gain due to hardware-aware single-node optimizations and hybrid OpenMP and MPI parallelization leads to speedups of 5–10 for typical production runs. The accomplished 5 seconds in simulation time for the Mount Merapi scenario show that the optimizations presented in this paper enable SeisSol to run not only scaling, but also production runs at petascale performance.

Considering the trend towards supercomputing architectures that are based on accelerators and many-core hardware, in general (GPUs, Intel MIC, e.g.), we demonstrated that substantial performance gains that address vectorization and efficient use of hybrid programming models also lead to substantial performance gains on commodity CPUs. In fact, the respective optimizations have prepared SeisSol for current and future many-core chips, such as Intel Xeon Phi [46].

Acknowledgements. We would like to thank Mikhail Smelyanskiy at Intel’s Parallel Computing Lab for assisting us in the development of hardware-aware kernels, as well as all colleagues of the Leibniz Supercomputing Centre who supported us in the execution of the large-scale SuperMUC runs.

Our project was supported by the Volkswagen Stiftung (project ASCETE: Advanced Simulation of Coupled Earthquake-Tsunami Events) and by the Bavarian Competence Network for Technical and Scientific High Performance Computing (KONWIHR). Computing resources for this project have been provided by the Leibniz Supercomputing Centre under grant: pr83no.

References

1. Breuer, A., Heinecke, A., Bader, M., Pelties, C.: Accelerating SeisSol by Generating Vectorized Code for Sparse Matrix Operators. In: International Conference on Parallel Computing (ParCo). Technische Universität München, Munich (2013)
2. Burstedde, C., Stadler, G., Alisic, L., Wilcox, L.C., Tan, E., Gurnis, M., Ghattas, O.: Large-scale adaptive mantle convection simulation. *Geophysical Journal International* 192(3), 889–906 (2013)
3. Carrington, L., Komatsch, D., Laurenzano, M., Tikir, M.M., Michéa, D., Le Goff, N., Snavely, A., Tromp, J.: High-frequency simulations of global seismic wave propagation using SPECFEM3D_GLOBE on 62K processors. In: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 60:1–60:11. IEEE Press, Austin (2008)
4. Cui, Y., Poyraz, E., Olsen, K., Zhou, J., Withers, K., Callaghan, S., Larkin, J., Guest, C., Choi, D., Chourasia, A., Shi, Z., Day, S.M., Maechling, J.P., Jordan, T.H.: Physics-based Seismic Hazard Analysis on Petascale Heterogeneous Supercomputers. In: International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE Press, Denver (2013)
5. Cui, Y., Olsen, K.B., Jordan, T.H., Lee, K., Zhou, J., Small, P., Roten, D., Ely, G., Panda, D.K., Chourasia, A., Levesque, J., Day, S.M., Maechling, P.: Scalable Earthquake Simulation on Petascale Supercomputers. In: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–20. IEEE Press, Washington, DC (2010)
6. Day, S.M., Bielak, J., Dreger, D., Graves, R., Larsen, S., Olsen, K., Pitarka, A.: Tests of 3D elastodynamic codes: Final report for Lifelines Project 1A02. Tech. rep., Pacific Earthquake Engineering Research Center (2003)
7. De La Puente, J., Käser, M., Dumbser, M., Igel, H.: An arbitrary high-order discontinuous Galerkin method for elastic waves on unstructured meshes—IV. Anisotropy. *Geophysical Journal International* 169(3), 1210–1228 (2007)
8. Dumbser, M., Käser, M.: An arbitrary high-order discontinuous Galerkin method for elastic waves on unstructured meshes – II. The three-dimensional isotropic case. *Geophysical Journal International* 167(1), 319–336 (2006)
9. Dumbser, M., Käser, M., Toro, E.F.: An arbitrary high-order Discontinuous Galerkin method for elastic waves on unstructured meshes—V. Local time stepping and p-adaptivity. *Geophysical Journal International* 171(2), 695–717 (2007)
10. Friedley, A., Bronevetsky, G., Lumsdaine, A., Hoefer, T.: Hybrid MPI: Efficient Message Passing for Multi-core Systems. In: International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE Press, Denver (2013)
11. Gerstenecker, C., Läufer, G., Steineck, D., Tiede, C., Wrobel, B.: Digital Elevation Models for Merapi. In: Microgravity at Merapi Volcano: Results of the First Two Campaigns, 1st Merapi-Galeras-Workshop (1999), DGG Special Issue
12. Gerstenecker, C., Läufer, G., Steineck, D., Tiede, C., Wrobel, B.: Validation of Digital Elevation Models around Merapi Volcano, Java, Indonesia. *Natural Hazards and Earth System Sciences* 5, 863–876 (2005)
13. Gmeiner, B., Köstler, H., Stürmer, M., Rüde, U.: Parallel multigrid on hierarchical hybrid grids: a performance study on current high performance computing clusters. *Concurrency and Computation: Practice and Experience* 26(1), 217–240 (2014)
14. Goto, K., Van De Geijn, R.: High-performance implementation of the level-3 BLAS. *ACM Transactions on Mathematical Software* 35, 1–14 (2008)

15. Graves, R., Jordan, T.H., Callaghan, S., Deelman, E., Field, E., Juve, G., Kesselman, C., Maechling, P., Mehta, G., Milner, K., Okaya, D., Small, P., Vahi, K.: CyberShake: A Physics-Based Seismic Hazard Model for Southern California. *Pure and Applied Geophysics* 168, 367–381 (2011)
16. Guest, M.: The Scientific Case for HPC in Europe 2012 - 2020. Tech. rep., Partnership for Advanced Computing in Europe (PRACE) (2012)
17. Heinecke, A., Breuer, A., Rettenberger, S., Bader, M., Gabriel, A., Pelties, C.: Optimized Kernels for large scale earthquake simulations with SeisSol, an unstructured ADER-DG code. In: International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE Press, Denver (2013) (poster abstract)
18. Iglberger, K., Hager, G., Treibig, J., Rude, U.: High performance smart expression template math libraries. In: International Conference on High Performance Computing and Simulation, pp. 367–373 (2012)
19. Intel Cooperation: Intel Math Kernel Library (Intel MKL) 11.0. Tech. rep., Intel Cooperation (2013), <http://software.intel.com/en-us/intel-mkl>
20. Karniadakis, G.E., Sherwin, S.J.: Spectral/hp Element Methods for Computational Fluid Dynamics. Oxford University Press (2007)
21. Karypis, G., Kumar, V.: Parallel Multilevel series k-Way Partitioning Scheme for Irregular Graphs. *SIAM Review* 41(2), 278–300 (1999)
22. Karypis, G., Kumar, V.: A Coarse-Grain Parallel Formulation of Multilevel k-way Graph Partitioning Algorithm. In: SIAM Conference on Parallel Processing for Scientific Computing. SIAM (1997)
23. Käser, M., Mai, P., Dumbser, M.: On the Accurate Treatment of Finite Source Rupture Models Using ADER-DG on Tetrahedral Meshes. *Bulletin of the Seismological Society of America* 97(5), 1570–1586 (2007)
24. Käser, M., Pelties, C., Castro, C., Djikpesse, H., Prange, M.: Wave field modeling in exploration seismology using the discontinuous galerkin finite element method on hpc-infrastructure. *The Leading Edge* 29, 76–85 (2010)
25. Käser, M., Dumbser, M., De La Puente, J., Igel, H.: An arbitrary high-order Discontinuous Galerkin method for elastic waves on unstructured meshes—III. Viscoelastic attenuation. *Geophysical Journal International* 168(1), 224–242 (2007)
26. Kramer, W.T.: Top500 Versus Sustained Performance: The Top Problems with the Top500 List – and What to Do About Them. In: 21st International Conference on Parallel Architectures and Compilation Techniques, pp. 223–230. ACM, New York (2012)
27. Krzikalla, O., Feldhoff, K., Müller-Pfefferkorn, R., Nagel, W.E.: Scout: A Source-to-Source Transformator for SIMD-Optimizations. In: Alexander, M., et al. (eds.) Euro-Par 2011, Part II. LNCS, vol. 7156, pp. 137–145. Springer, Heidelberg (2012)
28. Lay, T., Aster, R., Forsyth, D., Romanowicz, B., Allen, R., Cormier, V., Wysession, M.E.: Seismological grand challenges in understanding Earth’s dynamic systems. Report to the National Science Foundation, IRIS Consortium 46 (2009)
29. Mellor-Crummey, J., Garvin, J.: Optimizing Sparse Matrix-Vector Product Computations Using Unroll and Jam. *Int. J. High Perform. Comput. Appl.* 18(2), 225–236 (2004)
30. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard, Version 3.0. Specification, Message Passing Interface Forum (2012)
31. Meuer, H., Strohmaier, E., Dongarra, J., Simon, H.: Top500 list (November 2013), <http://www.top500.org>
32. Nishtala, R., Vuduc, R., Demmel, J., Yelick, K.: When cache blocking of sparse matrix vector multiply works and why. *Applicable Algebra in Engineering, Communication and Computing* 18(3), 297–311 (2007)

33. Oliker, L., Canning, A., Carter, J., Iancu, C., Lijewski, M., Kamil, S., Shalf, J., Shan, H., Strohmaier, E., Ethier, S., Goodale, T.: Scientific Application Performance on Candidate PetaScale Platforms. In: IEEE International Parallel and Distributed Processing Symposium, IPDPS 2007, pp. 1–12 (2007)
34. Olsen, K.B., Day, S.M., Minster, J.B., Cui, Y., Chourasia, A., Faerman, M., Moore, R., Maechling, P., Jordan, T.: Strong shaking in Los Angeles expected from southern San Andreas earthquake. *Geophysical Research Letters* 33(7) (2006)
35. OpenMP Architecture Review Board: OpenMP Application Program Interface Version 4.0 (2013), <http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf>
36. Pelties, C., Gabriel, A.A., Ampuero, J.P.: Verification of an ADER-DG method for complex dynamic rupture problems. *Geoscientific Model Development Discussion* 6, 5981–6034 (2013)
37. Pelties, C., Huang, Y., Ampuero, J.P.: Pulse-like rupture induced by three-dimensional fault zone flower structures. *Journal of Geophysical Research: Solid Earth* (to be submitted, 2014)
38. Pelties, C., Käser, M., Hermann, V., Castro, C.E.: Regular versus irregular meshing for complicated models and their effect on synthetic seismograms. *Geophysical Journal International* 183(2), 1031–1051 (2010)
39. Pelties, C., la Puente, J.D., Ampuero, J.P., Brietzke, G.B., Käser, M.: Three-Dimensional Dynamic Rupture Simulation with a High-order Discontinuous Galerkin Method on Unstructured Tetrahedral Meshes. *Journal of Geophysical Research: Solid Earth* 117 (2012)
40. Peter, D., Komatitsch, D., Luo, Y., Martin, R., Le Goff, N., Casarotti, E., Le Loher, P., Magnoni, F., Liu, Q., Blitz, C., Nissen-Meyer, T., Basini, P., Tromp, J.: Forward and adjoint simulations of seismic wave propagation on fully unstructured hexahedral meshes. *Geophys. J. Int.* 186(2), 721–739 (2011)
41. Pham, D.N., Igel, H., de la Puente, J., Käser, M., Schoenberg, M.A.: Rotational motions in homogeneous anisotropic elastic media. *Geophysics* 75(5), D47–D56 (2010)
42. Rabenseifner, R., Hager, G., Jost, G.: Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. In: 17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, pp. 427–436. IEEE Press, Washington, DC (2009)
43. Rabenseifner, R., Wellein, G.: Comparison of Parallel Programming Models on Clusters of SMP Nodes. In: Modeling, Simulation and Optimization of Complex Processes, pp. 409–425. Springer, Heidelberg (2005)
44. Rew, R., Davis, G.: NetCDF: an interface for scientific data access. *IEEE Computer Graphics and Applications* 10(4), 76–82 (1990)
45. Southern California Earthquake Center: 2014 Science Collaboration Plan. Tech. rep. (2013)
46. Vaidyanathan, K., Pamnany, K., Kalamkar, D.D., Heinecke, A., Smelyanskiy, M., Park, J., Kim, D., Shet, G.A., Kaul, B., Jo'o, B., Dubey, P.: Improving Communication Performance and Scalability of Native Applications on Intel Xeon Phi Coprocessor Clusters. In: 28th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2014. Phoenix (2014) (accepted for publication)
47. Virieux, J., Calandra, H., Plessix, R.D.: A review of the spectral, pseudo-spectral, finite-difference and finite-element modelling techniques for geophysical imaging. *Geophysical Prospecting* 59(5), 794–813 (2011)

48. Vuduc, R., Demmel, J.W., Yelick, K.A.: OSKI: A library of automatically tuned sparse matrix kernels. In: Scientific Discovery through Advanced Computing. Journal of Physics: Conference Series. Institute of Physics Publishing, San Francisco (2005)
49. Vuduc, R.W., Moon, H.J.: Fast Sparse Matrix-vector Multiplication by Exploiting Variable Block Structure. In: Yang, L.T., Rana, O.F., Di Martino, B., Dongarra, J. (eds.) HPCC 2005. LNCS, vol. 3726, pp. 807–816. Springer, Heidelberg (2005)
50. Wenk, S., Pelties, C., Igel, H., Käser, M.: Regional wave propagation using the discontinuous Galerkin method. *Journal of Geophysical Research: Solid Earth* 4(1), 43–57 (2013)
51. Whaley, R.C., Dongarra, J.J.: Automatically Tuned Linear Algebra Software. In: Conference on Supercomputing, pp. 1–27. IEEE Press, Washington (1998)
52. Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated empirical optimizations of software and the ATLAS project. *Parallel Computing* 27(1-2), 3–35 (2001)
53. Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., Demmel, J.: Optimization of Sparse Matrix-vector Multiplication on Emerging Multicore Platforms. In: International Conference for High Performance Computing, Networking, Storage and Analysis, New York, pp. 38:1–38:12 (2007)
54. Van Zee, F.G., van de Geijn, R.A.: BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Transactions on Mathematical Software* (2013) (accepted pending minor modifications)
55. Van Zee, F.G., Smith, T., Igual, F.D., Smelyanskiy, M., Zhang, X., Kistler, M., Austel, V., Gunnels, J., Low, T.M., Marker, B., Killough, L., van de Geijn, R.A.: The BLIS Framework: Experiments in Portability. *ACM Transactions on Mathematical Software* (2013) (accepted pending modifications)

SNAP: Strong Scaling High Fidelity Molecular Dynamics Simulations on Leadership-Class Computing Platforms

Christian R. Trott, Simon D. Hammond, and Aidan P. Thompson

Center for Computing Research,
Sandia National Laboratories, Albuquerque NM 87185, USA
`crtrott@sandia.gov`

Abstract. The rapidly improving compute capability of contemporary processors and accelerators is providing the opportunity for significant increases in the accuracy and fidelity of scientific calculations. In this paper we present performance studies of a new molecular dynamics (MD) potential called SNAP.

The SNAP potential has shown great promise in accurately reproducing physics and chemistry not described by simpler potentials. We have developed new algorithms to exploit high single-node concurrency provided by three different classes of machine: the Titan GPU-based system operated by Oak Ridge National Laboratory, the combined Sequoia and Vulcan BlueGene/Q machines located at Lawrence Livermore National Laboratory, and the large-scale Intel Sandy Bridge system, Chama, located at Sandia.

Our analysis focuses on strong scaling experiments with approximately 246,000 atoms over the range 1–122,880 nodes on Sequoia/Vulcan and 40–18,630 nodes on Titan. We compare these machine in terms of both simulation rate and power efficiency. We find that node performance correlates with power consumption across the range of machines, except for the case of extreme strong scaling, where more powerful compute nodes show greater efficiency.

This study is a unique assessment of a challenging, scientifically relevant calculation running on several of the world’s leading contemporary production supercomputing platforms.

1 Introduction

Classical molecular dynamics simulation (MD) is a powerful approach for describing the mechanical, chemical, and thermodynamic behavior of solid and fluid materials in a rigorous manner [5]. The material is modeled as a large collection of point masses (atoms) whose motion is tracked by integrating the classical equations of motion to obtain the positions and velocities of the atoms at a large number of timesteps. The forces on the atoms are specified by an inter-atomic potential that defines the potential energy of the system as a function of the atom positions. Typical inter-atomic potentials are computationally

inexpensive and capture the basic physics of electron-mediated atomic interactions of important classes of materials, such as molecular liquids and crystalline metals. Efficient MD codes running on commodity workstations are commonly used to simulate systems with $N = 10^5 - 10^6$ atoms, the scale at which many interesting physical and chemical phenomena emerge. For a few select physics applications, much larger atom counts are required, and these applications have historically been successfully run on leadership platforms [3,6,7,11]. Quantum molecular dynamics (QMD) is a much more computationally intensive method for solving a similar physics problem [9]. Instead of assuming a fixed interatomic potential, the forces on atoms are obtained by explicitly solving the quantum electronic structure of the valence electrons at each timestep. Because MD potentials are short-ranged, the computational complexity of MD generally scales as $O(N)$, whereas QMD calculations require global self-consistent convergence of the electronic structure, whose computational cost is $O(N_e^\alpha)$, where $\alpha = 2 - 3$ and N_e is the number of electrons. For the same reasons, MD is amenable to spatial decomposition on parallel computers, while QMD calculations allow only limited parallelism.

As a result, while high accuracy QMD simulations have supplanted MD in the range $N = 10 - 100$ atoms, QMD is still intractable for $N > 1000$, even using the largest supercomputers. Conversely, typical MD potentials often exhibit behavior that is inconsistent with QMD simulations. This has led to great interest in the development of MD potentials that match the QMD results for small systems, but can still be scaled to the interesting regime $N = 10^5 - 10^6$ atoms. These quantum-accurate potentials require many more floating point operations per atom compared to conventional potentials, but they are still short-ranged. So the computational cost remains $O(N)$, but with a larger algorithm pre-factor. This presents both opportunities and challenges in the context of Petascale computing. On the one hand, achieving good single-node performance is more difficult; on the other hand, the high compute-to-communication ratio implies enhanced strong scaling, relative to simpler potentials. In recent years Quantum Chemistry methods with $O(N)$ scaling have been developed. Similar to the quantum-accurate classical potentials, they allow the use of leadership class platforms for small to medium sized problems [4,13]. Nevertheless, the algorithm pre-factor in these methods is much larger than that of quantum-accurate potentials, making their use for MD simulation infeasible.

In this paper, we focus our attention on a new quantum-accurate potential called SNAP. It has been used to model the shear-migration of screw dislocations in tantalum metal, the fundamental process underlying plastic deformation in body-centered cubic metals [12]. Unlike simpler potentials the SNAP potential for tantalum correctly matches QMD results for the screw dislocation core structure and minimum energy pathway for displacement of this structure. Combining the accuracy of SNAP with efficient parallel algorithms that work on Petascale computers has opened the door to truly predictive first principles simulations of materials behavior at an unprecedented scale. The contributions in this paper are: (1) an analysis of the available parallelism and optimization opportunities for

the SNAP computational kernel; (2) description of thread-scalable implementation strategies for SNAP which achieves high performance on conventional multi-core CPUs, energy-optimized highly-threaded processors, and high-performance compute-oriented GPUs; (3) demonstration of SNAP with an unprecedented series of strong scaling simulations on leadership class supercomputing platforms with machine comparisons of both simulation rate and energy efficiency.

2 Mathematical Formulation of SNAP

The spectral neighbor analysis potential, or SNAP, is based on the common assumption that energy depends only the local arrangement of atoms in the material. The idea is to describe the local environment of each atom as an expansion of its neighbor density in spherical harmonic functions, and use a combination of the expansion coefficients as the energy contribution associated with that atom. Typically such an expansion uses the familiar basis of spherical harmonic functions $Y_m^l(\theta, \phi)$ multiplied with a separate radial part. Bartok *et al.* [2] instead chose to map the radial distance r to a third polar angle θ_0 and use 4D hyper-spherical harmonic functions $U_{m,m'}^j(\theta_0, \theta, \phi)$ as the basis for the expansion of the neighbor density of one atom:

$$\rho(\mathbf{r}) = \sum_{j=0, \frac{1}{2}, \dots}^{\infty} \sum_{m=-j}^j \sum_{m'=-j}^j u_{m,m'}^j U_{m,m'}^j(\theta_0, \theta, \phi) \quad (1)$$

The neighbor density of a central atom i in a particular configuration of atoms is written as a weighted sum of δ -functions. The sum is over all neighbor atoms i' within a cutoff distance R_{cut} :

$$\rho_i(\mathbf{r}) = \delta(\mathbf{0}) + \sum_{r_{ii'} < R_{cut}} f(r) w_{i'} \delta(\mathbf{r} - \mathbf{r}_{ii'}) \quad (2)$$

Here $\mathbf{r}_{ii'}$ is the 3D vector joining the position of the atoms i and i' . The $w_{i'}$ coefficients are dimensionless weights that are chosen to distinguish atoms of different chemical elements, while the central atom is arbitrarily assigned a unit weight. The function $f(r)$ ensures that the contribution of each neighbor atom goes smoothly to zero at R_{cut} . Using this formulation, each expansion coefficient can be expressed as a discrete sum over the neighbors:

$$u_{m,m'}^j = U_{m,m'}^j(0, 0, 0) + \sum_{r_{ii'} < R_{cut}} f(r_{ii'}) w_{i'} U_{m,m'}^j(\theta_0, \theta, \phi) \quad (3)$$

The coefficients $u_{m,m'}^j$ are complex and not invariant under rotation. Instead of using them directly in an expression for the energy, the so-called bispectrum components $B_{j_1,j_2,j}$ are used which are calculated as:

$$B_{j_1,j_2,j} = \sum_{m_1, m'_1 = -j_1}^{j_1} \sum_{m_2, m'_2 = -j_2}^{j_2} \sum_{m, m' = -j}^j (u_{m,m'}^j)^* H_{j_1, m_1, m'_1, j_2, m_2, m'_2}^{j, m, m'} u_{m_1, m'_1}^{j_1} u_{m_2, m'_2}^{j_2} \quad (4)$$

The constants $H_{j_1, m_1, m'_1, j_2, m_2, m'_2}^{j, m, m'}$ are known coupling coefficients, analogous to the Clebsch-Gordan coefficients for rotations on the 2-sphere. The bispectrum components $B_{j_1, j_2, j}$ are invariant under rotation and real valued. For later usage it is convenient to now also define the partial sums

$$Z_{j_1, j_2, j}^{m, m'} = \sum_{m_1, m'_1 = -j_1}^{j_1} \sum_{m_2, m'_2 = -j_2}^{j_2} H_{j_1, m_1, m'_1, j_2, m_2, m'_2}^{j, m, m'} u_{m_1, m'_1}^{j_1} u_{m_2, m'_2}^{j_2}. \quad (5)$$

The total energy due to the SNAP potential is composed of the N atom energies. The energy of each atom i is written as a linear sum of the K bispectrum components

$$E_{SNAP}^i = \sum_{i=1}^N E_{SNAP}^i = \beta_0^{\alpha_i} + \sum_{k=1}^K \beta_k^{\alpha_i} B_k^i, \quad (6)$$

where α_i indicates the chemical element identity of atom i . The β_k^α are the SNAP linear coefficients for atoms of type α , which are obtained by weighted least-squares linear regression against a large set of quantum calculations. Hence the problem of generating the inter-atomic potential has been reduced to that of choosing the best values for the SNAP linear coefficients. The SNAP force on an atom i can be expressed as a local sum over neighbors:

$$\mathbf{F}_{SNAP}^i = -\nabla_i E_{SNAP} = -\sum_{r_{ii'} < R_{cut}} \sum_{k=1}^K \beta_k^{\alpha_{i'}} \frac{\partial B_k^{i'}}{\partial \mathbf{r}_i}. \quad (7)$$

3 Force Algorithm and Parallelism in SNAP

The force on each atom due to the SNAP potential is formally given by Equation 7. In order to perform this calculation efficiently, we use a neighbor list, as is standard practice in the LAMMPS code [8,10]. This list identifies all the neighbors i' of a given atom i . In order to avoid negative and half-integer indices, we have switched notation from $u_{m, m'}^j$ to $u_{\mu, \mu'}^\eta$, where $\eta = 2j$, $\mu = m + j$, and $\mu' = m' + j$. Analogous transformations are used for $H_{j_1, m_1, m'_1, j_2, m_2, m'_2}^{j, m, m'}$ and $B_{j_1, j_2, j}$. Also, from this point on we identify the neighbor atom index with j instead of i' . Finally, boldface symbols with omitted indices such as \mathbf{u}_i are used to indicate a finite multidimensional array of the corresponding indexed variable.

Figure 1 gives the resulting force computation algorithm, where `calc_U(i)` calculates all expansion coefficients $u_{\mu, \mu'}^\eta$ from (3) for an atom i while `calc_Z(i, ui)` determines all $Z_{\eta_1, \eta_2, \eta}^{\mu, \mu'}$ as defined in (5). In the inner-loop first the derivatives of \mathbf{u}_i with respect to the distance vector between atoms i and j are computed and then the derivatives of B^i . The most computational expansive part of the algorithm is `calc_dBdR(i, j)` as given in Fig. 2. For the parameter sets used in this study, it is responsible for approximately 90% of all floating point and memory operations. Thus, we concentrate our description on this function.

```

Compute_SNAP() {
    for  $i$  in natoms() {
         $\mathbf{u}_i$  = calc_U( $i$ )
         $\mathbf{Z}_i$  = calc_Z( $i, \mathbf{u}_i$ )
        for  $j$  in neighbors( $i$ ) {
             $\nabla_j \mathbf{u}_i$  = calc_dUdR( $i, j, \mathbf{u}_i$ )
             $\nabla_j \mathbf{B}^i$  = calc_dBdR( $i, j, \mathbf{u}_i, \mathbf{Z}_i, \nabla_j \mathbf{u}_i$ )
             $\mathbf{F}_{ij}$  =  $-\beta \cdot \nabla_j \mathbf{B}^i$ 
             $\mathbf{F}_i$  +=  $-\mathbf{F}_{ij}$ ;  $\mathbf{F}_j$  +=  $\mathbf{F}_{ij}$ 
    } } }

```

Fig. 1. Base algorithm for the SNAP force calculation

```

Function Calc_dBdR( $i, j$ ) {
    for  $(\eta, \eta_1, \eta_2)$  in GetBispectrumIndices() {
         $\nabla_j B_{\eta_1, \eta_2, \eta} = 0$ 
        for  $(\mu = 0; \mu \leq \eta; \mu++)$  {
            for  $(\mu' = 0; \mu' \leq \eta; \mu'++)$  {
                 $\nabla_j Z_{\eta_1, \eta_2, \eta}^{\mu, \mu'} = 0$ 
                for  $(\mu_1 = \max(0, \mu + (\eta_1 - \eta_2 - \eta)/2);$ 
                     $\mu_1 \leq \min(\eta_1, \mu + (\eta_1 + \eta_2 - \eta)/2); \mu_1++)$  {
                     $\mu_2 = \mu - \mu_1$ 
                    for  $(\mu'_1 = \max(0, \mu' + (\eta_1 - \eta_2 - \eta)/2);$ 
                         $\mu'_1 \leq \min(\eta_1, \mu' + (\eta_1 + \eta_2 - \eta)/2); \mu'_1++)$  {
                         $\mu'_2 = \mu' - \mu'_1$ 
                         $\nabla_j Z_{\eta_1, \eta_2, \eta}^{\mu, \mu'} += H_{\eta_1, \mu_1, \mu'_1, \eta_2, \mu_2, \mu'_2}^{n, \mu, \mu'} (u_{\mu_1, \mu'_1}^{\eta_1} \nabla_j u_{\mu_2, \mu'_2}^{\eta_2} + u_{\mu_2, \mu'_2}^{\eta_2} \nabla_j u_{\mu_1, \mu'_1}^{\eta_1})$ 
                    } }
                     $\nabla_j B_{\eta_1, \eta_2, \eta} += (u_{\mu, \mu'}^{\eta})^* \nabla_j Z_{\eta_1, \eta_2, \eta}^{\mu, \mu'} + Z_{\eta_1, \eta_2, \eta}^{\mu, \mu'} (\nabla_j u_{\mu, \mu'}^{\eta})^*$ 
    } } } }

```

Fig. 2. Algorithm for the calculation of the bispectrum component derivatives. Typically more than 90% of all floating point operations are due to this function.

From these two algorithms it is clear that the overall structure of the force calculation is a seven-dimensional loop with the innermost loop performing several complex number calculations. Fortunately these loops provide a high degree of parallelism.

The outermost level is a loop over the atoms in a system. Part of that parallelism is used for domain decomposition on an MPI level, but each MPI rank will typically still handle multiple (and in many classic large scale simulations even millions) atoms. For the bulk of scientifically relevant simulations the number of atoms in a system lies in the range of $10^5 - 10^6$ atoms. The SNAP force calculation for each central atom i is completely independent of other atoms, assuming that current valid positions of all atoms have been distributed. In homogeneous systems, the total computational work for each atom is approximately the same.

The next loop is over the number of neighbors j of each atom i , calculating the force \mathbf{F}_{ij} due to the dependence of E_{SNAP}^i on the positions of i and j .

Table 1. Available parallelism for the SNAP potential with the parameter sets for silicon and tantalum atoms used in this study. Assumed is a system size of 100,000 atoms and the number of unique sets of loop indices corresponding to algorithms is given for the full system, for an average atom, and for one i, j interaction.

Sets	System	Atom	Interaction	Workload Variation
i	1.0×10^5	N/A	N/A	Low
i, j	1.6×10^6	15.5	N/A	None
$i, j, \eta_1, \eta_2, \eta$	1.0×10^8	992	64	Large
$i, j, \eta_1, \eta_2, \eta, \mu, \mu'$	2.1×10^9	32,528	1,388	Large
$i, j, \eta_1, \eta_2, \eta, \mu, \mu', \mu_1, \mu'_1$	2.4×10^{10}	2.4×10^5	15,183	None

For the parameter sets of SNAP so far developed at Sandia (including the ones used for the later benchmark runs) the average number of neighbors per atom is 14.6. For the class of materials targeted by SNAP it is expected that typical numbers are in the range of 15-50 neighbors per atom. Calculations on different pairs of atoms i, j are again independent except for a reduction at the end to accumulate force contributions into \mathbf{F}_i and \mathbf{F}_j . Note that interactions with the same central atom i require the same sets of \mathbf{u}_i and \mathbf{Z}_i . The computational work for each interaction is exactly the same for the single chemical element systems considered in this paper.

Within the function `Calc_dBdR(i, j)` the first loop is over the number of bispectrum components. For the parameter sets considered here there are 64 coefficients. As was the case on the two outer loops, calculations of different bispectrum components are essentially independent. After `Calc_dBdR` is finished a reduction over the bispectrum components takes place to compute \mathbf{F}_{ij} . In contrast to the previous two loops the work for each bispectrum component can be vastly different due to the complicated dependence of the loop boundaries for μ , μ' , μ_1 , and μ'_1 on the specific values of η_1 , η_2 , and η .

The next two loops over μ and μ' perform a reduction to calculate $\nabla_j B_{\eta_1, \eta_2, \eta}$, where for each pair μ and μ' a different $\nabla_j Z_{\eta_1, \eta_2, \eta}^{\mu, \mu'}$ is calculated through the two inner loops again as a reduction. The total number of unique sets $(\eta_1, \eta_2, \eta, \mu, \mu')$ is 1,388 for the SNAP parameter sets used in this study, while the total number of executions of the innermost loop body is 15,183 for each interaction. With the number of floating point operations in the innermost loop being 52, a single interaction requires on the order of 10^6 operations.

Table 1 provides a summary of the available parallelism. Depending on the targeted machine architecture different levels of parallelism have to be exploited. For CPU based architectures, large computer systems have on the order of 10^6 hardware threads. The largest such supercomputer is Sequoia, an IBM BG/Q system installed at the Lawrence Livermore National Laboratories in the United States, with roughly 100,000 nodes. Each node has 16 compute cores with 4 hyperthreads. Combined, this provides about 6×10^6 threads. Systems based on many-core architectures, such as the current Top500 leader Tianhe supercomputer, provide on the order of 10^7 threads. GPU based systems require even higher concurrency. With the number of threads needed per GPU on the order

Table 2. Relevant hardware parameters of the platforms used for strong scaling experiments. Data was obtained from the November 2013 Top500 list [1].

System	Nodes	Cores	Threads	Power/Node	Perf./Node
Chama	1,230	19,680	39,360	368.8 W	332.8 GFLOP/s
Sequoia/Vulcan	122,880	2.0×10^6	7.9×10^6	80.26 W	204.8 GFLOP/s
Titan	18,688	5.0×10^7	5.4×10^8	439.3 W	1450.8 GFLOP/s

of 50,000, large installations such as Titan require a total concurrency on the order of 10^9 .

Comparing these numbers with the available parallelism in a typical MD simulation using SNAP, a reasonable estimate is that one needs to parallelize over interactions (i, j pairs) on CPU and many-core based systems, while GPU systems require exposure of at least one, potentially two more levels.

In the following we discuss implementation details for CPU and GPU systems. In both cases we use domain decomposition to parallelize over nodes. All interactions of a single atom are handled by its assigned node (i.e. we do not distribute work of the same atom to multiple nodes).

4 CPU Implementation

As discussed in the previous section, it is necessary to parallelize over interactions in order to use large CPU based installations efficiently for the system sizes we want to consider. Parallelizing over interactions has a large appeal, since each interaction requires exactly the same amount of computation. As a consequence, a static work partitioning of interactions to threads is sufficient. Furthermore, relatively good load balancing can be expected even if the number of interactions is very small. The worst case scenario is that there is one more interaction to be computed on a node, than there are hardware threads.

In order to parallelize over interactions the loops over the atoms i and its neighbors j have to be flattened. Doing this poses the question of how to handle the calculation of \mathbf{u}_i and \mathbf{Z}_i . We implemented two different approaches: the *shared data* algorithm pulls the calculation of those arrays in front, and stores them for all atoms i in a globally accessible array. The *private data* algorithm handles these arrays as thread-private data. They must be computed by each thread individually each time it encounters a new central atom i . This causes duplicated calculations if multiple threads work on interactions of the same atom (i.e. $N_i \lesssim N_{\text{threads}}$). The *shared data* algorithm on the other hand requires much more memory. Together the arrays take about 300 kB per particle¹, which means that the whole data set will not fit into the last level cache of typical CPUs if there

¹ The \mathbf{Z}_i array is a five dimensional array whose dimension depends on the chosen order of the spherical harmonic density expansion. For the parameter sets used in this study the dimension is 7. Since \mathbf{Z}_i are complex values the total amount of data is thus $7^5 * 2 * 8$ byte $\simeq 300kB$. The \mathbf{u}_i array is only three-dimensional and does significantly affect the total amount of needed memory.

<i>shared data</i> algorithm	<i>thread-private data</i> algorithm
<pre> Compute_SNAP() { for i in natoms() { $\mathbf{u}[i] = \text{calc_U}(i)$ $\mathbf{Z}[i] = \text{calc_Z}(i, \mathbf{u}[i])$ } make_list_of_IJ_pairs() parallel_for i, j in pairs() { $\nabla_j \mathbf{u}_i = \text{calc_dUdR}(i, j, \mathbf{u}[i])$ $\nabla_j \mathbf{B}^i = \text{calc_dBdR}(i, j, \mathbf{u}[i], \mathbf{Z}[i], \nabla_j \mathbf{u}_i)$ $\mathbf{F}_{ij} = -\beta \cdot \nabla_j \mathbf{B}^i$ $\mathbf{F}_i += -\mathbf{F}_{ij}; \mathbf{F}_j += \mathbf{F}_{ij}$ } } </pre>	<pre> Compute_SNAP() { make_list_of_IJ_pairs() parallel_for i, j in pairs() { if($i \neq i_{old}$) { $\mathbf{u}_i = \text{calc_U}(i)$ $\mathbf{Z}_i = \text{calc_Z}(i, \mathbf{u}_i)$ } $\nabla_j \mathbf{u}_i = \text{calc_dUdR}(i, j, \mathbf{u}_i)$ $\nabla_j \mathbf{B}^i = \text{calc_dBdR}(i, j, \mathbf{u}_i, \mathbf{Z}_i, \nabla_j \mathbf{u}_i)$ $\mathbf{F}_{ij} = -\beta \cdot \nabla_j \mathbf{B}^i$ $\mathbf{F}_i += -\mathbf{F}_{ij}; \mathbf{F}_j += \mathbf{F}_{ij}$ $i_{old} = i$ } } </pre>

Fig. 3. The *shared data* and *thread-private data* algorithm of the SNAP force calculation

are significantly more than 100 atoms per node. Fortunately, the *private data* algorithm performs well in that situation, since interactions of the same central atom i will usually be handled by the same thread, eliminating the duplication of calculations.

Figure 4 shows the performance of SNAP running on a dual socket Intel Sandy Bridge CPU system with varying number of atoms (blue line). 32 threads have been used (*i.e.* hyperthreading is enabled). Performance is given in GFLOP/s which is calculated from runtimes using algorithmic floating-point operations and average neighbor counts. This means any redundant implementation calculations for \mathbf{u}_i are not included. For 64 or more atoms the *private data* algorithm has been used. The performance increases with atom count, going from 21 GFLOP/s at 2 atoms to a peak of 47 GFLOPS/s at 64 atoms, which corresponds to 30 interactions per thread. This curve sets the limit for what can be expected in a strong scaling experiment with multiple nodes. When one applies strong scaling until each node has only 2 atoms, parallel efficiency can not be expected to be higher than 37%.

It is worth noting that an efficiency of 37% with only two atoms on a node represents a vast improvement in scalability over commonly used MD simulations. For simple potentials such as Lennard-Jones and EAM, the efficiency at that point would be less than 1%. A parallel efficiency of 37% is also high enough that it is not unreasonable to run large scale simulations with as little as 2 atoms per node. With a simulation rate of 700 timesteps/s this would allow for typical simulation times of a few million timesteps to finish in a matter of hours.

5 GPU Implementation

For the GPU implementation using CUDA more parallelism than just the total number of interactions is needed. As a first approach we utilized the *shared-data* approach with thread blocks of 64 threads handling an interaction. Each

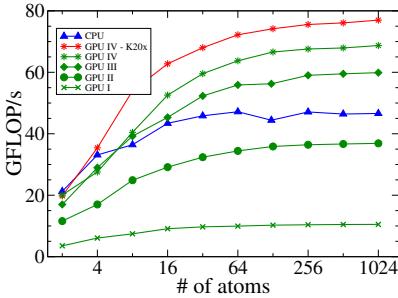


Fig. 4. Performance of SNAP with varying number of atoms on a workstation with dual Intel Sandy Bridge CPUs and NVIDIA GPUs

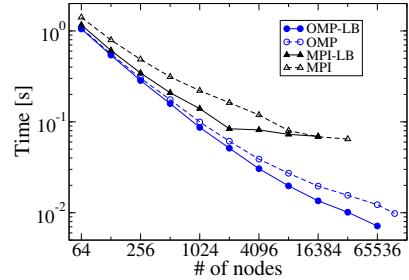


Fig. 5. Comparison of MPI-only and MPI+OpenMP runs with both micro load-balancing enabled and disabled

thread within a thread block performs the calculation for exactly one bispectrum component. All temporary data including the per-interaction data is kept in global memory, since the amount of available shared-memory (48 kB) would only suffice for a single block with 64 threads. This data is read through the texture cache, mitigating the performance penalty of the irregular access pattern.

Figure 4 shows that the performance achieved with this initial implementation (GPU-I) is approximately four times lower than that of the previously discussed CPU implementation. This might have been expected considering that this first implementation ignores the fact that for each bispectrum component a different amount of work has to be performed. As a consequence strong load imbalances of threads within the same block occur, resulting in many cores being idle for most of the time. In order to estimate how strong this effect is we recorded how many innermost loop executions (work items) happen for each bispectrum component. The bispectrum component with the most work executes the innermost loop 1,369 times, which is 6 times higher than the average number of executions of 237 times and approximately 9% of the total number of work items of 15,183.

In order to address this load imbalance it is necessary to expose a finer level of parallelism. There are 1,388 sets of $(i, j, \eta_1, \eta_2, \eta, \mu, \mu')$ indices with the highest number of work items for a single set being 49. By grouping multiple sets together it is possible to organize the work in a way that each thread in a thread block of up to 320 threads gets approximately 50 work items assigned, resulting in good load balancing. To facilitate this the algorithm is rewritten to flatten out the loops of Calc_dBdR into a single super-loop. The complete 7 index sets are stored as an array of structs, with each thread looping over a subset of the complete list. Since each index only goes from 0 to 7 it was possible to encode the struct into a single 32 bit integer through bit masks. Coefficients representing a 63rd order expansion of the density could be encoded in a 64 bit integer. By sorting the list appropriately it is possible to make sure that the innermost reduction to compute $\nabla_j Z_{\eta_1, \eta_2, \eta}^{\mu, \mu'}$ is handled by a single thread, so that the sum can be performed in a local variable without atomic updates. $\nabla_j B_{\eta_1, \eta_2, \eta}$ on the other hand is updated atomically.

Since with this approach more threads work on a single interaction (up to 320 instead of 64), shared memory can now be used for the temporary data without restricting the number of active threads. To further increase the number of active threads this temporary data is also stored in single precision only. Using this approach it is possible to fit three active blocks with a total of almost 1,000 threads on a single GPU multi-core, which is high enough to provide the on GPUs necessary latency hiding. Consequently, performance goes up by a factor of approximately three versus the first implementation, achieving 75% of the CPU performance.

A detailed profiling analysis shows that the biggest remaining problem are bank conflicts in shared memory. Looking at the access pattern also reveals that less than half of the entries in the $7 \times 7 \times 7$ temporary arrays are ever accessed. Indeed given the indices k, l , and m it holds true that $l \leq k$ and $m \leq k$. Consequently only $\sum_{i=0}^7 i * i = 140$ will ever be accessed. Using the partial sums it is possible to calculate compressed offsets into the temporary arrays, thus reducing the amount of necessary storage and making accesses more dense. The latter actually reduces the number of bank conflicts. The compression also allows the implementation to return to using full double precision for the temporary data, without reducing the number of threads per GPU multi-core. On top of this, a further reduction in bank conflicts can be achieved by padding the arrays. Since the access pattern is irregular a brute force method was employed to determine the most effective padding size, which turned out to be 159. While this size is specific to the current parameter set using a 7th order expansion, it is trivial to expand the concept and simply determine for each expansion order the most effective padding. This could even been done using autotuning. In combination these two measures improved performance by 50% over the second implementation, making the GPU faster than the CPU variant.

A further improvement is achieved by merging calc_dUdR and calc_dBdR into a single kernel allowing the first part to generate temporary data directly into shared memory instead of putting it into global memory with the calc_dBdR kernel loading it back into shared memory. One issue here is that calc_dUdR does not expose as much parallelism as calc_dBdR. Only 32 threads of a block are taking part in that calculation, and it was necessary to exploit instruction level parallelism to keep those threads busy.

We also experimented with the number of threads by making the buckets for each thread larger or smaller. Experiments showed that 320 threads per block (the maximum number while still being able to distribute roughly equal work to each thread) is not the optimal number, and we used 288 instead. At that point resources on a SM were used almost optimally. The total number of used registers reaches 62k out of 64k and 47.5 kB of the available 48 kB shared memory are utilized.

Resulting improvements, denoted as GPU-IV, added 15% to the previous implementation, reaching 1.47x of the CPU performance. Also shown in Fig. 4 are results with a NVIDIA K20x GPU which is slightly higher performing than the previously used NVIDIA K20c GPU. The latter is a workstation product with

active cooling, while the former is intended for server installations and is found in most large clusters including Titan. The K20x reaches a peak performance of 77 GFLOP/s, or 168% of the peak performance of the dual Sandy Bridge CPUs. Exposing all this parallelism enables a respectable 25% efficiency with only 2 atoms on a single GPU. Similar to the CPU implementation about 64 atoms per GPU are required to achieve full performance.

Note that for both CPU and GPU performance is most likely limited by the irregular memory access in the innermost loop of calc_dBdR. There are about 2.5 floating point operations per 8 byte memory load. This means that the K20x provides an effective bandwidth of about 250 GB/s, while the CPU provides an effective bandwidth of about 150 GB/s. Those numbers are only possible for irregular memory access since the work sets are small enough that virtually all accesses are serviced by cache.

6 Scaling Studies

Scaling studies were performed on three systems: Chama, Sequoia/Vulcan, and Titan all of which are listed in the November 2013 Top500 list (Chama as # 100, Sequoia as # 3, Vulcan as # 9 and Titan as # 2). The same code described with the CPU-algorithm was used on Sequoia/Vulcan and on Chama. Note that in early 2012 Sequoia and Vulcan, which are actually two partitions of one large BG/Q installation at the Lawrence Livermore National Laboratories, were available as a single system. Our scaling studies were one of the few successfully performed experiments on the entire installation. We believe that the strong scaling results we achieved on this system scaling from a single node to all 122,880 nodes are unique.

For the type of simulations targeted with SNAP, load balancing between MPI processes is a strong requirement. Even when simulating dense materials, small local density fluctuations occur. When trying to scale to single atoms per MPI rank, these small density fluctuations can lead to huge load imbalances. While LAMMPS has load balancing mechanisms, those are targeted at a much coarser level. Essentially LAMMPS decomposes a simulation box with planes in three dimensions. For load-balancing purposes LAMMPS can move those planes independently. This is an approach which works very well for gradual density changes in a system. For the type of fluctuations which pose a problem in our case, this type of load balancing does not work.

To remedy this we implemented a micro load-balancer. Instead of changing domain boundaries it reassigned responsibility of calculating the force for individual atoms. Essentially an MPI rank with more than the average number of atoms, checks all neighboring domains until it finds one with less than the average number of atoms. The load balancer then transfers responsibility for a single atom to that neighbor MPI rank. Consequently a maximum of 26 atoms can be given away, or received in a single load balancing pass. A major cost of this approach is that generally the halo regions have to be twice as large, so that a MPI rank which receives responsibility to calculate the force of an atom is guaranteed to know about the positions of its neighbor atoms. Early experiments

have shown that on GPU based systems this cost is larger than the potential time savings. On CPU based systems on the other hand a large positive effect can be observed.

Figure 5 shows the result of a strong-scaling experiment on Sequoia with 65,536 atoms comparing MPI-only and MPI+OpenMP runs with both micro load-balancing enabled and disabled. For the MPI-only runs 64 MPI ranks per node were used and two per node with 32 threads each for the MPI+OpenMP runs. Using two MPI ranks per node instead of one for the threaded runs enables better utilization of the network resources. Furthermore this means that even with a single atom per MPI rank most threads will actually have an interaction to work on.

First it is obvious that the threaded implementation achieves much better performance than the non-threaded one even for small node counts. Considering the reduction in communication this is not surprising. Also the fastest runtime achieved by the MPI-only runs is 64 ms per timestep as compared to 7 ms per timestep for the OpenMP run with load-balancing. At that point only one in 32 cores actually performs force calculations in the MPI-only runs.

Activating load-balancing improves the runtime of the MPI-only runs. Even with as little as 64 nodes performance increases by 22%. At that point an MPI process has only 8 atoms on average, so small fluctuations in density are noticeable. A detailed analysis showed that the load-balancer achieved uniform distribution of atoms over the MPI ranks. As a consequence, the performance curve is much steeper, reaching a sharp inflection point at 2,048 nodes, where the load-balancer has allocated one atom to every second MPI rank. Interestingly, the inflection point is not at 1,024 nodes, where each MPI rank has 1 atom. This is because on average only two of the four MPI ranks allocated to each processor core have an atom to work on. Each active rank therefore has more hardware resources available. As a result, the performance is improved by up to 1.93x for the MPI-only case, and by up to 1.70x for the hybrid MPI and OpenMP case.

In order to compare different architectures we run a strong scaling experiment of a system with 245,760 atoms on the three large scale clusters available to us: Chama, Sequoia/Vulcan and Titan. The particular system size was chosen because it represents a typical small to medium sized system used in many scientific studies, and it allows us to go to the limit of scaling on Sequoia/Vulcan with 2 atoms per node (1 atom per MPI rank) at full scale. We ran the same system size on the other machines to enable direct comparisons. We show both traditional measures, as well as the power normalized curves because core count or nodes can be a misleading metric to base a comparison on. A Sequoia/Vulcan node for example only uses about 80 Watts and has a theoretical peak of about 200 GFLOP/s, while a Titan node requires about 440 Watts and has a theoretical peak of almost 1500 GFLOP/s. Thus we tie the comparison to energy, one of the two biggest constraints determining super-computer design today (cost, the other major constraint, is hard to quantify and dependent on many soft factors such as exchange rates, rebates inflation etc.). All numbers are based on the data published in the November 2013 Top500 list [1]. Note that the power

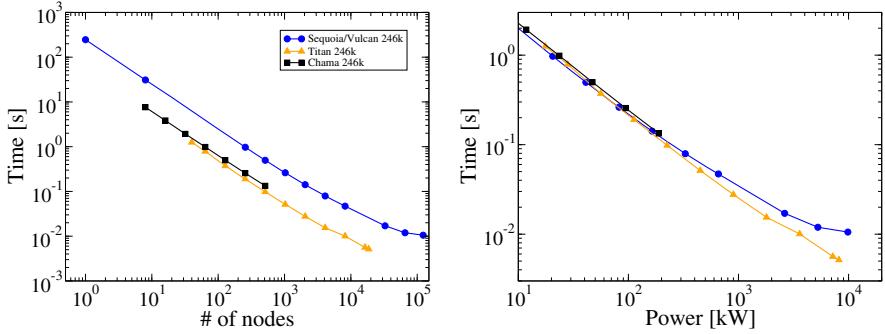


Fig. 6. Time for performing a single simulation step with 246k atoms on Sequoia/Vulcan, Titan and Chama. On the right the curves have been scaled by power per node.

consumption reported in the Top500 list is obtained when running the HPL LINPACK benchmark, which shares characteristics of our SNAP potential i.e. floating-point dense calculations utilizing a small working set. The node power consumption as well as theoretical peak performance are calculated by taking total system values and dividing by the number of nodes. For power consumption this means that cooling and network energy costs are effectively included in the node values. For Sequoia/Vulcan we used the power per node data from Sequoia. All values are listed in Tab. 2.

Figure 6 shows the time per timestep for a strong scaling run with 245,760 atoms. In the left panel time is plotted against number of nodes, while in the right panel the curves are scaled by power consumption per node. When scaling out to full system size Titan is about two times faster than Sequoia/Vulcan. Most of this factor is due to Titan having more atoms per node at full scale (~ 13 versus 2 on Sequoia/Vulcan). Thus the surface to volume ratio from the domain decomposition is better and the GPUs are operating in a range where they can still be filled effectively. This is reflected by the parallel efficiencies which drop only to about 50% on Titan compared with 14% on Sequoia/Vulcan. Chama is not large enough to show any significant loss of parallel efficiency for a system of 246k atoms. Normalizing the performance to power consumption as shown on the right in Fig. 6 makes the relative energy efficiency much clearer. For small to medium node counts all three systems are with 20% of each other with respect to energy efficiency. In this regime Sequoia/Vulcan is actually the most effective one followed by Titan and then Chama. Only at larger node counts (i.e. less work per node) Titan is becoming more efficient than Sequoia/Vulcan. The crossover point is reached at about 200 atoms per GPU.

In Fig. 7 normalized performance in GFLOP/s per node is shown with respect to a normalized workload in atoms per node in the left panel, as well as the power normalized curves in the right panel. To get the latter the curves were scaled by power per node on both axes. In the figure three more runs with smaller sizes chosen to reach the limit of scaling on the systems are added to the 246k atom runs. Note how the curves for each machine overlap, which means that

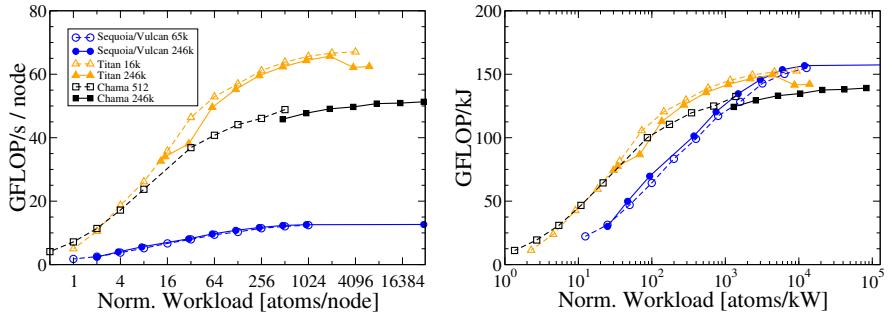


Fig. 7. Normalized performance plotted against normalized workload. In the right panel values have been scaled by power per node. Two strong scaling runs with different number of atoms are shown for each system.

performance per node is mainly determined by the number of atoms per node and largely independent of the total number of atoms, i.e. very good weak scaling is achieved. While on a per node basis BG/Q is dramatically slower than Chama or Titan, it is as efficient as the other systems for large workloads when normalizing by power consumption. For medium and small workloads Chama and Titan are up to three times as power efficient, since work is much less spread out in those systems, reducing the total amount of communication necessary. Note that in terms of simulation rate Titan achieves its peak performance at 2 atoms/GPU (roughly 4 atoms/kW). Chama can achieve even higher simulations rate by going to a single atom per node, or ~ 1.5 atoms/kW.

7 Conclusion

SNAP is a novel, high-fidelity approach to performing scientifically relevant atomistic simulations at the intermediate scale of 10^5 to 10^6 atoms. In this paper we have presented implementations of SNAP optimized for a range of high-performance computing hardware from contemporary multi-core processors to new energy optimized architectures such as highly threaded CPUs and compute-oriented GPUs. An efficient micro load-balancing scheme is also presented which allows strong scaling down to a single atom per MPI rank. We demonstrate that by performing micro load-balanced, strong scaled simulations we are able to utilize the entirety of the Titan and Sequoia/Vulcan supercomputers - some of the largest leadership platforms currently available. The experiences obtained in developing such an algorithm are a sentinel for many of the issues which algorithm developers may expect to face at Exascale, particularly that such systems can efficiently execute higher fidelity algorithms for contemporary physics applications where simply increasing the problem size may not be scientifically appropriate. Finally, we compare the energy efficiency of these architectures showing a strong correlation between performance and energy

except in the limit of extreme strong scaling where more powerful compute nodes, such at those on Titan, deliver higher energy efficiency.

Acknowledgement. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

Furthermore we are grateful for access to compute resources at the Lawrence Livermore National Laboratory and support from Livermore Computing both of which are supported by the U.S. Department of Energy under contract DE-AC52-07NA27344.

References

1. TOP500 Supercomputer Site, <http://www.top500.org>
2. Bartok, A.P., Payne, M.C., Risi, K., Csanyi, G.: Gaussian Approximation Potentials: the Accuracy of Quantum Mechanics, without the Electrons. *Physical Review Letters* 104, 136403 (2010)
3. Glosli, J.N., Richards, D.F., Caspersen, K.J., Rudd, R.E., Gunnels, J.A., Streitz, F.H.: Extending Stability Beyond CPU Millennium: A Micron-scale Atomistic Simulation of Kelvin-Helmholtz Instability. In: *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC 2007, pp. 58:1–58:11. ACM, New York (2007)
4. Goedecker, G.: Linear Scaling Electronic Structure Methods. *Rev. Mod. Phys.* 71, 1085 (1999)
5. Griebel, M., Knapek, S., Zumbusch, G.: Numerical Simulation in Molecular Dynamics. Springer, Heidelberg (2007)
6. Hamada, T., Narumi, T., Yokota, R., Yasuoka, K., Nitadori, K., Taiji, M.: 42 TFlops Hierarchical N-body Simulations on GPUs with Applications in Both Astrophysics and Turbulence. In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC 2009, pp. 62:1–62:12. ACM, New York (2009)
7. Kadau, K., Germann, T.C., Lomdahl, P.S.: Molecular Dynamics Comes of Age: 320 Billion Atom Simulation on BlueGene/L. *International Journal of Modern Physics C* 17(12), 1755–1761 (2006)
8. LAMMPS. LAMMPS molecular dynamics package WWW site, lammps.sandia.gov, Potential benchmarks, lammps.sandia.gov/bench.html
9. Mattsson, A.E., Schultz, P.A., Desjarlais, M.P., Mattsson, T.R., Leung, K.: Designing Meaningful Density Functional Theory Calculations in Material Science—A Primer. *Modelling Simul. Mater. Sci. Eng.* 31, R1–R31 (2005)
10. Plimpton, S.: Fast Parallel Algorithms For Sort-Range Molecular-Dynamics. *J. Comput. Phys.* 117(1), 1–19 (1995)

11. Swaminarayan, S., Germann, T.C., Kadau, K., Fossum, G.C.: 369 Tflop/s Molecular Dynamics Simulations on the Roadrunner General-Purpose Heterogeneous Supercomputer, pp. 1–10 (November 2008)
12. Thompson, A.P., Swiler, L.P., Trott, C.R., Foiles, S.M., Tucker, G.: A new Quantum-Accurate Interatomic Potential for Tantalum (2014) (in preparation)
13. Wang, L.-W., Lee, B., Shan, H., Zhao, Z., Meza, J., Strohmaier, E., Bailey, D.H.: Linearly scaling 3d fragment method for large-scale electronic structure calculations. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing. SC 2008, pp. 65:1–65:10. IEEE Press, Piscataway (2008)

Exascale Radio Astronomy: Can We Ride the Technology Wave?^{*}

Erik Vermij¹, Leandro Fiorin¹, Christoph Hagleitner², and Koen Bertels³

¹ IBM Research, The Netherlands

² IBM Research – Zurich, Switzerland

³ Delft University of Technology, Delft, The Netherlands

erik.vermij@nl.ibm.com

Abstract. The Square Kilometre Array (SKA) will be the most sensitive radio telescope in the world. This unprecedented sensitivity will be achieved by combining and analyzing signals from 262,144 antennas and 350 dishes at a raw datarate of petabits per second. The processing pipeline to create useful astronomical data will require exa-operations per second, at a very limited power budget. We analyze the compute, memory and bandwidth requirements for the key algorithms used in the SKA. By studying their implementation on existing platforms, we show that most algorithms have properties that map inefficiently on current hardware, such as a low compute-bandwidth ratio and complex arithmetic. In addition, we estimate the power breakdown on CPUs and GPUs, analyze the cache behavior on CPUs, and discuss possible improvements. This work is complemented with an analysis of supercomputer trends, which demonstrates that current efforts to use commercial off-the-shelf accelerators results in a two to three times smaller improvement in compute capabilities and power efficiency than custom built machines. We conclude that waiting for new technology to arrive will not give us the instruments currently planned in 2018: one or two orders of magnitude better power efficiency and compute capabilities are required. Novel hardware and system architectures, to match the needs and features of this unique project, must be developed.

1 Introduction

The Square Kilometre Array (SKA) [1] will be the biggest radio telescope in the world, and will have an unprecedented sensitivity, angular resolution and survey speed. Most specifications are ten to a 100 times better than any existing telescope. Because of the size of the project, its construction has been divided into two phases: SKA1 and SKA2. SKA1 is currently being designed, and construction will start in 2018. In the same year, the design of SKA2 will start. This paper will only look at the SKA1, because the specifications for SKA2

* This work is conducted in the context of the joint ASTRON and IBM DOME project and is funded the Netherlands Organization for Scientific Research (NWO), the Dutch Ministry of EL&I, and the Province of Drenthe, The Netherlands.

have yet to be finalized. SKA1 will deploy 262,144 antennas and 350 dishes in remote areas in South-Africa and Australia, together producing several petabits of data per second. Realizing the SKA1 will face many challenges in diverse fields like data transport, algorithms, data storage and system design. In this paper we will look at the computational challenges of the project: the absolute performance and power efficiency required. Power efficiency has special attention, since several subsystems of the SKA1 will be located far away from any human infrastructure.

The main contributions of this paper are as follows. We present a detailed computational profile of the SKA1 and its main algorithms, analyze the algorithms on existing hardware, and discuss points for improvement. We show relevant trends in high-performance computing and introduce the innovation metric to compare generations of supercomputers. Finally, we argue about the feasibility of deploying the SKA1 using commercial off-the-shelf (COTS) hardware.

2 SKA1 Project Description

SKA1 [2] will consist of three instruments: SKA1-low, SKA1-mid and SKA1-survey. In this paper, we will foremost analyze SKA1-low, with excursions to the other two instruments where needed and appropriate.

SKA1-low is an aperture array instrument [3] consisting of 1024 stations, each containing 256 dual-polarized antennas, which will receive signals between 50 and 350 MHz. The antenna signals are summed per station into a single beam, which is transported to a central signal-processing facility. The stations will be 35 m in diameter, and spaced over a distance of 70 km. This instrument will be very much like a big version of LOFAR, the low-frequency aperture array built in the Netherlands [4].

SKA1-mid will use 254 single-pixel feed dishes capable of receiving signals between 350 MHz and 13.8 GHz. From this frequency range, a 2.5 GHz band can be selected for measurements. The dishes will be placed at most 200 km apart.

The SKA1-survey instrument will use 96 dishes, each containing phased array receivers. Every receiver will have 94 antennas and can point 36 beams onto the sky. In this way, a single dish has a huge field of view, compared with the SKA1-mid dishes. The frequency ranges between 250 MHz and 4 GHz, with an instantaneous bandwidth of 500 MHz. The antennas will be spaced at most 50 km apart.

SKA1 will serve many science cases. In this work, we only focus on the continuum imaging science case, which creates sky images for the entire frequency range.

2.1 Processing Pipeline and Algorithms

In Figure 1, we show the simplified processing flow for the SKA1-low, using the continuum imaging science case. For the SKA1-mid and SKA1-survey, the antenna/station subsystem will be replaced by a dish.

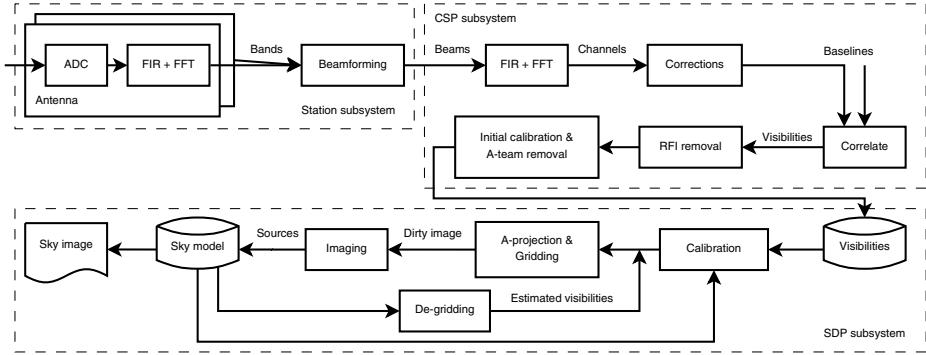


Fig. 1. Overview of the processing steps for the SKA1-low, using the continuum imaging science case. For SKA1-mid and SKA1-survey, the antenna/station subsystem will be replaced by a dish.

Station Processing for the SKA1-low. Because the SKA1-low uses aperture arrays instead of dishes, extra processing is required to synthesize a dish, as explained hereafter. The digitized antenna signal from the analog-digital converter is sent to a polyphase channelizer consisting of several finite impulse response (FIR) filter banks and a fast Fourier transform (FFT), creating a number of frequency bands. These are fed into the beamformer, in which every band is multiplied with a complex phase shift to delay the signal, and added to the corresponding band from the other antennas. By delaying signals between antennas and summing them, the instrument focuses its sensitivity into a specific direction, creating a so-called beam. The channelization before beamforming is needed because the beamforming concept only works on small frequency bandwidths [5].

Central Signal Processing (CSP). The beam from every station or dish undergoes a second channelization, generating frequency channels, after which the beams are aligned in time and phase and, in case of the SKA1-low, undergo a gain correction to offset filter artifacts from the station processing. The beam is correlated (multiplied) [6] with the data from all other stations or dishes, and integrated over a small period of time (the dump time). By correlating, the signal-to-noise ratio of the data improves. A pair of stations/dishes is called a baseline. The result is a visibility, which is a sample of the Fourier-transformed sky. The visibilities are processed by removing RFI signals and by performing a calibration step, to correct for known system inequalities. Furthermore, a set of well-known very bright sources (the A-team) is demixed from the dataset. After these steps, the data is often integrated again in time and frequency, depending on the frequency smearing [7] and/or other science requirements.

Science Data Processor (SDP). From the visibilities, a sky image can be constructed. The calibration works on a station/dish basis, and accounts for

direction-dependent effects (DDEs), such as ionospheric distortion, beam shape and others. The result is a parameter set which is passed to the A-projection algorithm [8]. From the parameters, we can create a small map representing the complex gain function for a beam (analog to the lens behavior in an optical camera). Two maps of a single baseline are multiplied together, and then multiplied with a W-term to account for the non-coplanar baselines effect [9] (the earth is not flat), and scaled up. The resulting map is multiplied with a visibility and added (gridded) onto a Fourier grid. A Fourier transform of the grid results in a dirty image, and the CLEAN imaging algorithm [10] is used to extract bright sky sources (imaging). After a certain threshold has been reached, the extracted sources are converted back into visibilities (de-gridding), which are subtracted from the original dataset. This gives us a visibilities dataset with only weak sources, and the gridding process starts over until only noise is left. All extracted sources are kept in a sky model. This sky model is used to make better estimations about certain calibration parameters, resulting in another feedback loop back into the calibration step. After a sufficient amount of iterations, the sky model can be converted into a sky image, ready for research.

3 SKA1 Computational Profile

Figure 2 shows a high-level estimate of the compute, bandwidth, and data in local storage required for some of the major algorithms, for the SKA1-low. The numbers are an extension of the work by Jongerius et al. [11]. We start with a very high input bandwidth, and over time we trade bandwidth into data products. The input bandwidth for the griddler is so high because of a great demand for metadata: the incoming visibilities are a fraction of the total. Two major bandwidth reduction points can be identified. First, when beamforming, data from all antennas are summed per station. Secondly, when gridding, we transform an incoming visibility stream into a Fourier image. Regarding compute requirements, there are three major peaks in the peta-operations per second range. These are the station channelization, correlation and A-projection/ gridding (where gridding dominates). As de-gridding is located in the feedback loop, it is omitted in the Figure for sake of clarity, but it has compute requirements similar to those of gridding. The other SKA1 instruments show a similar compute distribution: heavy requirements for the correlator, A-projection and gridding.

3.1 Analysis of Key Algorithms

In this work, we focus in particular on four algorithms, which are the most demanding from the computational point of view, and should run as efficiently as possible in order to make an impact on SKA1. A summary of their properties is shown in Table 1. Together they require over 90% of the compute operations for the SKA1.

To cope with the size of the system, we divide it into independent compute units (ICUs). These units, also shown in Table 1, can be executed completely

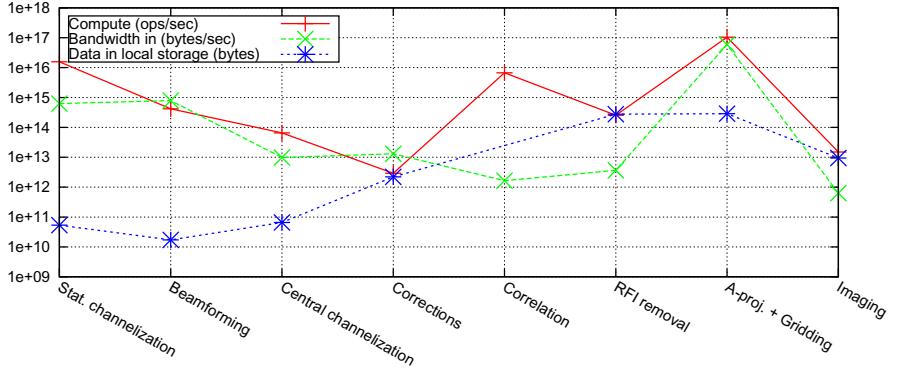


Fig. 2. A high-level estimate of the compute, bandwidth, and data in local storage required for the key algorithms of SKA1-low. The units for the Y-axis are in the legend.

independently at a system level. The terminology correspond with the labels in Figure 1.

Station Channelization. The channelization consists of a fixed-size bank of FIR filters and a fixed-size (real to complex) FFT. The compute requirements per second per ICU are given in Equation 1. N_{taps} is the number of filter taps (equal to 5) and N_{bands} is the number of frequency bands (equal to 1024). The input data rate, $Samples_{sec}$, is equal to 800 mega sample per second. Incoming samples are 8 bits wide, whereas outgoing samples are 16-24 bits wide.

$$Ops_{channelization} = Samples_{sec} \times (2N_{taps} + 5 \times 0.5\log_2(2N_{bands})) \quad (1)$$

Table 1. Typical numbers for several metrics of the key algorithms per ICU, for the SKA1-low. ICUs are Independent Compute Units, as described in Section 3.1.

	Station channelization	Correlator	A-projection	Gridding
ICU	Antenna	Channel	Channel/ Baseline	Channel
Number of ICUs	524288	262144	274877906944 / S _{chan}	524288
Compute (op/sec)	30 giga	25.6 giga	136 mega	188 giga
I/O (bytes/sec)	800/ 2400 mega	6.25/ 20.83 mega	*/ 576 kilo	113 giga/ *
Compute/ IO (op/byte)	10.11	945.23 (in theory)	9.8	14 [12]
Mem. access pattern	Scatter gather	Line	Scatter gather	Line
Mem. address generation	Static	Configurable	Flexible	Flexible
Instruction generation	Static	Configurable	Flexible	Flexible

Correlator. The correlator multiplies two signals together and adds the result to a sum. The compute for the correlator per second per ICU is given in Equation 2. Here, N_{stat} is the number of stations or dishes. The input rate $Samples_{sec}$ for the SKA1-low is just over 3 kilo samples per second per station per ICU. Incoming samples are 8 bits wide, outgoing samples up to 32 bits.

$$Ops_{correlator} = 8 \times Samples_{sec} \times 0.5 \times N_{stat}^2 \quad (2)$$

An implementation challenge lies in the fact that data arrives per station, containing all the frequency channels of that station, whereas the algorithm wants the data per frequency channel, containing all the stations. This data rearrangement is often called the ‘cornerturn’, and frustrates practical implementations of the correlator.

A-projection. The A-projection algorithm consists of several 2D FFTs and point-wise matrix multiplications. The compute is however dominated by a single 2D FFT, often not a power of two in size. The compute requirements per second per ICU are given by Equation 3. Here, W is the average W-matrix size (on average 30 for SKA1-low), and O is a scaling factor (equal to 8). S_{sec} indicates how many seconds the projection matrices are valid because of time dependent effects (equal to 10-120 sec). S_{chan} indicates how many frequency channels can be served by the same projection matrices (equal to 700 channels). N_{iter} is the number of iterations following the feedback loops described in Section 2.1, which is on the order of 30. Unless stated otherwise, the values given are based on research for LOFAR.

$$Ops_{A\text{-projection}} = 5 \times \frac{N_{iter}}{S_{sec} S_{chan}} \times W^2 O^2 \log(W^2 O^2) \quad (3)$$

The up-scaling introduces a form of interpolation in the gridded. This is necessary because the location of the visibilities is of much higher precision than the Fourier plane gridpoints.

Gridding. As de-gridding and gridding are very much the inverse of each other with the same kernels and properties, we will only focus on gridding in this paragraph. Based on the location of the visibility with respect to the grid, a 1/64th subset of the matrix produced by the A-projection algorithm is selected. The visibility is multiplied with this matrix and added to the Fourier plane. The compute per second per ICU is given in Equation 4. T_{dump} is the correlator dump time (0.6 sec for SKA1-low), and N_{bl} the number of baselines. The datawidth and datatype for this algorithm can be from 16 up to 64 bits and either fixed or floating point.

$$Ops_{gridding} = (6 + 2) \times \frac{N_{iter} N_{bl}}{T_{dump}} \times W^2 \quad (4)$$

A practical aspect of this algorithm is that additional parallelism in the baselines exists. However, exploiting that can be difficult because of the addition to a final grid, which has to happen in an atomic way.

4 The SKA1 on Today’s Technology

As shown in the preceding sections, SKA1 will require significant computational power. In this section, we will be analyzing state-of-the-art implementations of the key algorithms, and take a look at how we could optimize current technology for SKA1.

4.1 Core Technologies

A decade ago, the first *multicore* CPU was introduced, which sacrificed single-core peak performance for parallelism. With *manycore* architectures, this concept is taken to the extreme. Besides using more cores, *specialized* logic designed for a specific task can be added: using additional area is traded off for better power efficiency or throughput/latency [13]. Performance and/or power efficiency can be further improved by using *heterogeneity*. This can be done on a node level by using multiple types of devices (CPUs and GPUs for example), or within a single device (ARM big.LITTLE [14] for example). Another kind of heterogeneity comes in the form of attached FPGAs, which are *reconfigurable*. Examples of this are the Molen polymorphic processor [15], and the systems of Convey Computer [16].

Most advances in power efficiency and throughput we see today are based on these technologies. For example, GPUs employ the manycore paradigm, have thousands of small cores, and are often used in a heterogeneous setup. CPUs become faster by adding wider and more specialized instructions [17] or small accelerators [18].

4.2 The SKA1 Algorithms on Existing Products

Station Channelization. The work performed by Shahbahrami et al. [19] shows that FIR filters using real numbers are well suited for SIMD parallelization. For complex numbers additional data-shuffling hardware is required. Jongerius et al. [20] show that a modern CPU can only run a real FIR filter at 10-15% of its peak performance, because of the low number of operations per byte of I/O. Romein [21] shows that this also holds true for complex FIR filters on GPUs.

The FFT algorithm features an irregular data access pattern with low computational intensity, which make it hard to run this algorithm efficiently on almost any architecture. Jongerius et al. [20] show that the FFTW library [22] reaches 17% of the peak performance of a modern CPU. The research by Xu et al. [23] shows in detail how the FFT can be optimized for SIMD processing on a modern CPU. Romein [21] shows that FFTs on GPUs are heavily IO bound, and

achieve around 20% of the peak performance, comparable with CPUs. Research by Lobeiras et al. [24] and nVidia’s own CUFFT library confirm this.

LOFAR shows us that using FPGAs on the Uniboard [25] is a good match for the fixed-sized FIR and FFT. The multiple bitwidth modes LOFAR can use are all supported on a single FPGA image.

For the channelization we can conclude that SIMD and SIMT models both work fine, and that the bottlenecks are in the memory bandwidth and memory access pattern.

Correlator. Nieuwpoort et al. [26] implemented the correlation algorithm on several architectures, namely CPUs, GPUs, the Blue Gene®/P [27], and the Cell processor. They conclude that having sufficient local storage is key to good performance. Both Cell and the Blue Gene® achieve close to peak performance, whereas CPUs and GPUs reach significantly lower numbers (70 and 40% respectively). Work by Clark et al. [28] and Romein et al. [21] shows that for modern GPUs, a peak performance of up to 80% can be achieved for a large number of stations, provided that significant optimization effort is put into register and IO usage.¹

Research by Woods [29] and Souza et al. [30] shows that FPGAs are a suitable candidate for running the correlation algorithm, because of its regular and simple structure. Utilizing custom datawidths is a major, and realistic, advantage over other architectures.

In conclusion, the correlator runs well on most architectures, but a big local storage is important. Using custom datatypes gives an advantage.

A-projection. There are no publications available that show the performance, utilization and/or power efficiency of this algorithm on hardware, as it is rather novel. Given that the algorithm is dominated by FFTs, the behavior on hardware be similar. As the size of the FFT is not fixed, and changes from baseline to baseline and channel to channel, using hardwired (FPGA) solutions will be less successful.

Here we can conclude the same as for the channelization, but given the flexible FFT size, more flexibility will be required.

Gridding. The naive way to implement this algorithm gives a very low computational intensity [31], and a lot of atomic add operations. This is shown in [32] for CPUs and GPUs. Both platforms achieve around 5% of their peak performance because of bandwidth limitations. An implementation on the Cell [33] achieves around 25% of the peak performance, but only after considerable optimizations in the memory bandwidth usage. GPU research performed in [12] tries to eliminate atomic add instructions as much as possible, and the GPU

¹ Blue Gene is a trademark of International Business Machines Corporation, registered in many jurisdictions worldwide. Other product or service names may be trademarks or service marks of IBM or other companies.

reaches 25% of its peak performance. Even when there are hardly any atomic add operations left (0.23% of all grid updates), they still take up 26% of the time.

For the gridding, we can conclude that the simple kernel will run well on most architectures; bottlenecks are in the memory bandwidth and atomic additions.

4.3 Hints towards Optimized Architectures

This section presents and analyzes two aspects of the key SKA1 algorithms: first, the power breakdown on CPUs and GPUs, and second, the cache and ALU performance on CPUs. Together with the work presented in Section 4.2, these analyses can be seen as a starting point for research into architectures that can run the algorithms at higher throughputs and with improved power efficiency. The CPU tests are run on a model of an Intel® Xeon® E5-2630 implemented by using the Gem5 simulator [34] and McPAT [35], while the GPU tests are run on a generic model of a NVIDIA® GTX480 GPU implemented by using an enhanced version of GPUWattch [36], a microarchitecture-level GPU power simulator based on GPGPU-Sim [37] and McPAT. This GPU architecture is not the most modern one, and already superseded by two architectures. The general organisation of GPUs has however not changed, therefore we believe the results shown are still relevant. The CPU performance numbers are obtained by using the Intel® Vtune™ Amplifier and again an Xeon® E5-2630 processor.

For the station channelization, we used the algorithm described by Romein [21]. The correlator implementations are based on code presented by Nieuwpoort et. al. [26], and the gridding implementation described by Romein [12]. For the A-projection algorithm, we implemented a 2D FFT algorithm processing as input a 128×128 complex matrix, which is a realistic number for the SKA1-low.

Power Breakdown. Figure 3 shows the results of our power breakdown experiments. As can be seen, the CPU shows very similar power distributions for all the algorithms. Only a very small portion of the energy goes to the actual computations. Around 50% of the energy goes to the three levels of caches, and another 20% to 30% is spent in the memory controllers. Although not shown in the graph, almost all the power for the level two and level three caches is due to the static power component. For the level one cache, the static and dynamic power distribution is about equal. This means that a significant part of a CPUs power usage is due to the presence of these big caches, which consume power even when they are idle. From the graph it is possible to notice the slightly higher usage of the memory controllers in the station channelization, due to the streaming nature of the algorithm. Moreover, the correlator uses more power in the level one cache, which corresponds with its intensive use of a local storage, as indicated Section 4.2.

The breakdown for the GPU shows a bigger power percentage for the functional units than the CPU, as one would expect with GPUs being compute oriented platforms. Another interesting observation is the large amount of energy GPUs spend in the register file (especially compared to CPUs), around

15%. This corresponds with the fact that the register file in a GPU is very big, and can be compared with the level one cache in CPUs. The constant power (around 15%) is mainly caused by processor and memory leakage power and peripheral circuits' power [36]. In the case of GPUs, it can be noticed that we have a high power consumption of the floating point units while running the correlator. Furthermore, the A-projection algorithm consumes a lot of energy in the local storage (register files, shared memory and level two cache), corresponding with the large datasets it has to in a relative inefficient way.

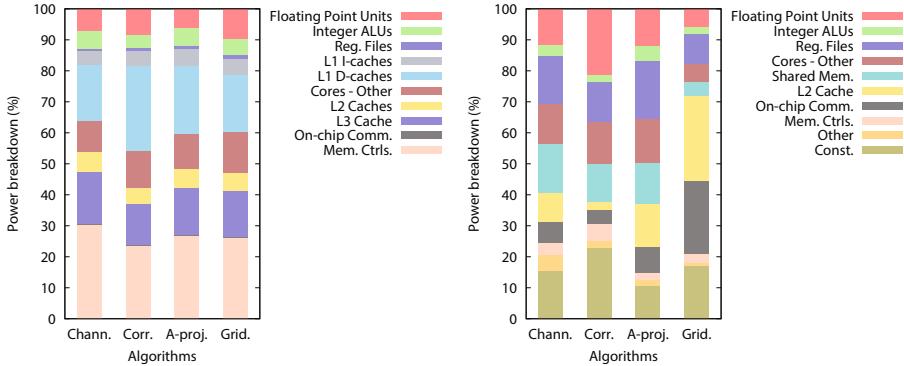


Fig. 3. Power breakdown on CPUs (left) and GPUs (right) for the four SKA1 algorithms considered in this work: station channelization (Chann.), correlator (Corr.), A-projection (A-proj.) and Gridding (Grid.).

CPU Cache Performance. The results for the CPU cache performance are shown in Table 2, and correspond with what we know about the algorithms. The station channelization does not use the level two and level three cache: data is streamed from external RAM into level one, processed and evicted again. For the correlator we know that it appreciates large amounts of local storage, this is reflected in the numbers: level one misses almost always hit in level two. The A-projection behaves in the same way as the station channelization, but also uses the level two cache, since it uses larger datasets. The gridding exhibits very good cache behavior. Regarding ALU utilization, only the correlator shows good performance. The station channelization and A-projection are limited by the FFT structure (an isolated kernel will not run at a higher than 50% utilization), and gridding is hampered by complex address calculations and inefficient loads.

Considerations. From these numbers, we can extract some general guidelines on how to improve the power efficiency and throughput of these four algorithms.

GPUs spend a significant amount of energy in the floating point units, the register file and the rest of the core. We see a similar picture for the corresponding CPU categories. Adding macro instructions to the core-architecture for executing FFT butterflies (beyond existing shuffle instructions), complex multiplications

Table 2. CPU cache performance and ALU utilization for the four SKA1 algorithms considered in this work. Here Chann. is the station channelization.

	Chann.	Correlator	A-projection	Gridding
Cycles under L1\$ miss	29%	32%	46%	1%
Cycles under L1\$ miss, L2\$ hit	3%	29%	26%	1%
Cycles under L2\$ miss, L3\$ hit	4%	2%	7%	0%
Cycles under L3\$ miss	23%	1%	13%	0%
ALU utilization	20%	64%	23%	34%

or even bigger blocks, would result in less register file accesses and more efficient execution units. This could reduce the power consumption and improve the throughput. Furthermore, core-architectures could be optimized to capture the simplicity and regularity of the algorithms. The correlator for example, has basically a single type of instruction (complex multiply accumulate) inside several loops, but still, when running this algorithm on a CPU, five percent of the power goes to the instruction cache. Exploiting this regularity can also improve the power efficiency and throughput.

Besides the computational part of the chips, the local memories use large amounts of energy, and are often also a performance bottleneck. Making the memories deeper and wider will make the performance better, but the energy consumption even worse. A possible improvement would be to be able to exploit the regularity and predictability of the data needs. With carefully arranged accesses, specific to our SKA1 algorithms, we could utilize the available bandwidth to the largest extend. Similarly an improved, more SKA1 specific, organization of the local memories would increase data locality, reducing the number of accesses to a lower level or the external memory. The numbers presented in Table 2 and the power breakdown observations in Section 4.3 can serve as a guideline here.

5 HPC Trends

In this section, we will look at several HPC trends, foremost based on the Top 500 [38] and Green 500 [39]. With the petaflops well achieved, the high-performance computing community is looking at the next big step: an exaflops of sustained Linpack performance. In this section, we use the data from the November 2013 lists.

Peak Performance Analysis. Figure 4 shows the measured performance of the number one systems of the Top 500 for the past decade. Note we only include new number one systems: including the number one from every list would make the Figure less clear, while not changing the result. It can be seen that on a logarithmic scale, this dataset fits a straight line, which goes back all the way back to 1993. The performance of the number one systems increases with roughly

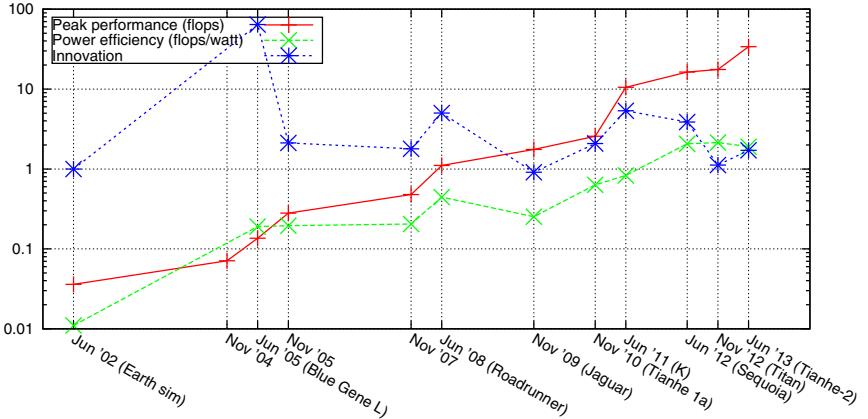


Fig. 4. Trends for the new number one systems in the Top500. Innovation is explained in Section 5. The units for the Y-axis are in the legend.

a factor of two every year. These performance projections suggest that it should be possible to build an exaflop system in 2018.

Power Efficiency Lags behind Peak Performance. Figure 4 shows, in flops per watt, the power efficiency of the system. Also in this case, we can observe a straight line. The slope of the power efficiency is lower than that of the peak performance, meaning that peak performance is growing exponentially faster than power efficiency. This means that new systems use exponentially more absolute power than their predecessors. We can generalize these Linpack numbers for general-purpose workloads [40]: power bills are becoming higher and cooling will become more impractical. When the practical absolute power usage limit is reached in the not so distant future, the growth in peak performance will have to slow down to follow the trend in power efficiency.

Innovation Analysis. Every new number one supercomputer is faster than its predecessor, and most of the time also more power efficient. We are interested in the level of innovation that every new generation brings, i.e., how much closer it brings us to high-performance and power-efficient computing. To investigate this, we create the Innovation metric, shown in Equation 5. $Perf$, Eff and n are the measured peak performance, power efficiency and system index, respectively. The results are shown in Figure 4. The higher this number the better.

$$Innovation_n = \frac{Perf_n}{Perf_{n-1}} \times \frac{Eff_n}{Max(Eff_0 : Eff_{n-1})} \quad (5)$$

Over the past decade, an interesting observation is that, out of all the systems, the best scoring ones are all based on custom-designed hardware (Blue Gene®/L, PowerXCell®, K, Blue Gene®/Q). The lower scoring systems are all based on

commercial off the shelf (COTS) products, or are extensions of existing systems. This analysis shows us that although the use of COTS products is convenient for many reasons, they apparently do not give us the big steps forward we need.

5.1 Heterogeneity and Power Efficiency in HPC

In recent years, accelerators have gained in popularity in supercomputers. These products are often celebrated for their power efficiency and peak performance.

For power efficiency, we look at the Green 500: the Top 500 sorted by power efficiency. The current, GPU-heavy, green number one system achieves an impressive 4.5 gigaflop per watt (1.9 for the Top 500 number one), but is a factor 220 slower in peak performance. This difference in peak performance makes comparing the efficiency hard, because small systems can always be more power efficient than big ones, because of less communication overhead.

The first big supercomputer in the green list is the Piz Daint. This system is number six in the Top 500 (6.2 petaflops), and number four in the Green 500 (3.18 gigaflops per watt). This shows that it is possible to build big heterogeneous supercomputers, with around 50% better performance per watt than the homogeneous Blue Gene®/Q systems. These results might, however, not hold for a wider set of benchmarks. This is reflected in [41], noting that Linpack no longer represents real workloads, which are often not as GPU-friendly as the DGEMM kernel in Linpack.

In the Top 500, only a little over 10% of the systems use accelerators, a number that hardly changed in the last three lists. Important factors for this might be the extra development effort they require, the narrower application space that will run well on them, and the only modest improvement in peak performance [42] and power efficiency as shown above.

6 Technology Challenges For the SKA1

6.1 Traditional Walls

The traditional technology walls have been power, memory and instruction-level parallelism. Dennard scaling [43] no longer works, thus with every technology scaling of S , the amount of transistors that can be switched on while staying within the power envelope, shrinks with a factor S [44]. Furthermore, moving data costs a lot of power: moving data from external RAM to the compute core can be a factor 1000 more expensive than doing a computation on that data [45].

Regarding memory, we see that compute performance is growing exponentially faster than bandwidth [46], and latency lags bandwidth quadratically [47]. This means that getting enough data to feed the cores on time is becoming increasingly hard. As an example, for NVIDIA's® Tesla® product line, the bandwidth-to-compute ratio has worsened by a factor 2.15 in the past 5 years (C870 to K40) [48].

Lastly, instruction-level parallelism is limited in every real application, thereby limiting the return on investments regarding wider execution machines [49].

6.2 SKA1 Challenges

For SKA1, the power and memory walls can be summarized into a data-locality problem. The instrument generates so much data, and requires often so few operations per byte (see Table 1 and Section 4.2), that it is impossible to move the data to an appropriate processing device. As a node level example: a modern PCIe-attached device needs hundreds of operations per incoming byte to run at a high utilization. Between nodes we see a similar problem: a significant portion of supercomputing power goes to the interconnect, and with the rapidly increasing node peak performance, data communication cannot keep up. Managing where the data is with respect to the compute resources will be the key challenge.

As discussed in the introduction, power efficiency is of special importance to the SKA1, since several subsystems will be located in remote areas. Power will have to be transported there or generated on the spot. This gives another dimension to the power efficiency challenge.

7 Can SKA1 Ride the Technology Wave?

7.1 At Node Level

The algorithms discussed and analyzed in Sections 3 and 4 exhibit features that do not map well on existing technologies. In Section 4.3, we already briefly discussed some of them: predictable data-access patterns, high bandwidth-to-compute ratio, and complex arithmetic. Other exploitable features are narrow and/or flexible datawidths and simple highly repetitive kernels.

Given the challenging requirements of the SKA1, we cannot afford to either waste computational resources or energy or not benefit from some specific features the algorithms offer us.

7.2 On System Level

In Figure 5, we show the compute versus power efficiency operation points for the several subsystems of SKA1. The SKA1 systems should be built towards the end of this decade. Power requirements are found in the SKA1 baseline design, and the compute is based on Jongerius et al. [11] and this work. The graph furthermore shows several point sets and a trend based on several Top 500 top ten systems. We would like to emphasize that our kernels will run on a lower utilization than the Linpack kernel for the Top/Green 500 datapoints.

Two observations can be made. First, the power efficiency of the subsystems are orders of magnitude apart. This means that the power budgets are not balanced very well. As illustration we also included a weighted power efficiency in Figure 5. Secondly, it is clear that riding the technology wave and waiting for systems available at the end of the decade will not give us the hardware needed to realize some subsystems (foremost the SDPs) of SKA1.

Given the data challenges indicated in Section 6, novel data-centric approaches should be considered. A high-level example of this could be to split the system

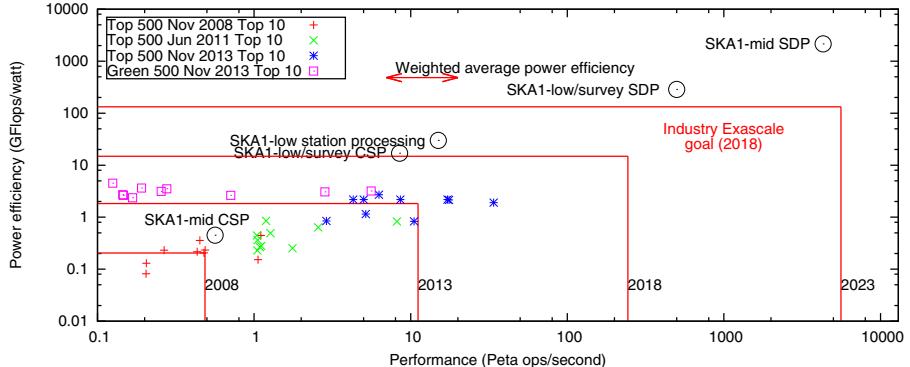


Fig. 5. The compute and power efficiency points for the SKA1 subsystems

physically into independent compute units. Data can stay at rest within an ICU, while the node does all consecutive processing steps. This would significantly reduce the load on the system interconnect, and thereby reduce the power consumption. With such an approach, we would end up with a true data-centric, heterogeneous supercomputer.

8 Conclusion – Realizing the SKA1

In this paper, we presented an overview of SKA1 from a compute perspective, and analyzed whether the instrument can be built using COTS hardware in 2018. Three observations can be made: 1) SKA1 will require large amounts of computational power at a very high power efficiency. 2) Most SKA1 algorithms will only run at efficiencies around 10% to 50% on COTS hardware, with clear points for improvement being smarter memory accesses and specialized execution units. 3) The HPC innovation and trend analysis shows that custom-build machines achieve two to three times larger improvement in compute capabilities and power efficiency than systems employing COTS hardware.

We conclude that waiting for new technology to arrive will not give us the instruments currently planned at the end of the decade: one or two orders of magnitude better power efficiency and compute capabilities are required. Developing new technology tailored for the specific tasks at hand must be considered. Solving the data-locality problem will be key: there should always be a suitable compute element near the data, on every level.

References

1. SKA: Square Kilometer Array, <http://www.skatelescope.org/>
2. SKA: SKA Baseline design (2013),
https://www.skatelescope.org/wp-content/uploads/2012/07/SKA-TEL-SK0-DD-001-1_BaselineDesign1.pdf

3. Perley, R.A.E.: A proposal for a large, low frequency array located at the VLA site. VLA Scientific Memorandum 146 (1984)
4. van Haarlem, M., Wise, M., Gunst, A., Heald, G., McKean, J., et al.: LOFAR: The LOw-Frequency ARray. *Astronomy & Astrophysics* (May 2013)
5. Jeffs, B.: Beamforming presentation,
<http://ens.ewi.tudelft.nl/Education/courses/et4235/Beamforming.pdf>
6. Thompson, A.R., Moran, J.M., Swenson, G.W.: *Interferometry and Synthesis in Radio Astronomy*, 2nd edn. Wiley-VCH, Weinheim (2001)
7. Bridle, A.H., Schwab, F.R.: Wide Field Imaging I: Bandwidth and Time-Average Smearing. *Synthesis Imaging in Radio Astronomy* 6, 247 (1989)
8. Tasse, C., van der Tol, B., van Zwieten, J., van Diepen, G., Bhatnagar, S.: Applying full polarization A-Projection to very wide field of view instruments: An imager for LOFAR. *Instrumentation and Methods for Astrophysics* (December 2012)
9. Cornwell, T., Golap, K., Bhatnagar, S.: The non-coplanar baselines effect in radio interferometry: The W-Projection algorithm. *IEEE Journal of Selected Topics in Signal Processing* 2 (2008)
10. Clark, B.G.: An efficient implementation of the algorithm ‘CLEAN’. *Astronomy and Astrophysics* 89, 377–378 (1980)
11. Jongerius, R., Wijnholds, S., Nijboer, R., Corporaal, H.: End-to-end compute model of the square kilometre array. *IEEE Computer* (accepted, 2014)
12. Romein, J.W.: An efficient work-distribution strategy for gridding radio-telescope data on gpus. In: ACM International Conference on Supercomputing (ICS 2012), Venice, Italy, pp. 321–330 (2012)
13. Venkatesh, G., Sampson, J., Goulding, N., Garcia, S., Bryksin, V., Lugo-Martinez, J., Swanson, S., Taylor, M.B.: Conservation cores: reducing the energy of mature computations. *SIGARCH Comput. Archit. News* 38, 205–218 (2010)
14. ARM: big.little,
<http://www.arm.com/products/processors/technologies/biglittleprocessing.php>
15. Vassiliadis, S., Wong, S., Gaydadjiev, G., Bertels, K., Kuzmanov, G., Panainte, E.: The MOLEN polymorphic processor. *IEEE Transactions on Computers* 53, 1363–1375 (2004)
16. Convey: Convey computer website, <http://www.conveycomputer.com>
17. Intel: Intel SSE and AVX extensions,
<http://software.intel.com/en-us/intel-isa-extensions>
18. Intel: Intel random number generator,
http://software.intel.com/sites/default/files/m/d/4/1/d/8/441_Intel_R__DRNG_Software_Implementation_Guide_final_Aug7.pdf
19. Shahbahrami, A., Juurlink, B., Vassiliadis, S.: Efficient vectorization of the FIR filter. In: Proc. 16th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC), pp. 432–437 (2005)
20. Jongerius, R., Corporaal, H., Broekema, C., Engbersen, T.: Analyzing LOFAR station processing on multi-core platforms. *ICT Open 2012* (2012)
21. Romein, J.: Signal Processing on GPUs for Radio Telescopes. In: GPU Technology Conference 2013 (2013)
22. Frigo, M., Johnson, S.G.: FFTW: An adaptive software architecture for the FFT. In: Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing, vol. 3, pp. 1381–1384. IEEE (1998)
23. Xu, W., Yan, Z., Shunying, D.: A high performance FFT library with single instruction multiple data (SIMD) architecture. In: International Conference on Electronics, Communications and Control (ICECC), pp. 630–633 (2011)

24. Lobeiras, J., Amor, M., Doallo, R.: FFT Implementation on a Streaming Architecture. In: 19th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp. 119–126 (2011)
25. Szomoru, A.: The UniBoard: A multi-purpose scalable high-performance computing platform for radio-astronomical applications. In: XXXth URSI General Assembly and Scientific Symposium, pp. 1–4 (2011)
26. Nieuwpoort, R., Romein, J.: Correlating radio astronomy signals with many-core hardware. *International Journal of Parallel Programming* 39, 88–114 (2011)
27. Romein, J.W., Broekema, P.C., Mol, J.D., van Nieuwpoort, R.V.: The LOFAR correlator: implementation and performance analysis. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2010, pp. 169–178. ACM, New York (2010)
28. Clark, M.A., Plante, P.C.L., Greenhill, L.J.: Accelerating Radio Astronomy Cross-Correlation with Graphics Processing Units. *CoRR* abs/1107.4264 (2011)
29. Woods, A.: Accelerating software radio astronomy fx correlation with gpu and fpga co-processors. Master’s thesis, University of Cape Town (2010)
30. de Souza, L., Bunton, J., Campbell-Wilson, D., Cappallo, R., Kincaid, B.: A Radio Astronomy Correlator Optimized for the Xilinx Virtex-4 SX FPGA. In: International Conference on Field Programmable Logic and Applications, FPL 2007, pp. 62–67 (2007)
31. van Amesfoort, A.S., Varbanescu, A.L., Sips, H.J., van Nieuwpoort, R.V.: Evaluating Multi-core Platforms for HPC Data-intensive Kernels. In: Proceedings of the 6th ACM Conference on Computing Frontiers, CF 2009, pp. 207–216. ACM, New York (2009)
32. Humphreys, B., Cornwell, T.: Analysis of convolutional resampling algorithm performance (2011),
http://www.skatelescope.org/uploaded/59116_132_Memo_Humphreys.pdf
33. Varbanescu, A.L., van Amesfoort, A.S., Cornwell, T., van Diepen, G., van Nieuwpoort, R., Elmegreen, B.G., Sips, H.: Building high-resolution sky images using the Cell/B.E. Sci. Program. 17, 113–134 (2009)
34. Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A., Hestness, J., Hower, D.R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M.D., Wood, D.A.: The gem5 simulator. SIGARCH Comput. Archit. News 39, 1–7 (2011)
35. Li, S., Ahn, J.H., Strong, R., Brockman, J., Tullsen, D., Jouppi, N.: McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In: MICRO-42. 42nd Annual IEEE/ACM International Symposium on Microarchitecture, pp. 469–480 (2009)
36. Leng, J., Hetherington, T., ElTantawy, A., Gilani, S., Kim, N.S., Aamodt, T.M., Reddi, V.J.: GPUWattch: Enabling Energy Optimizations in GPGPUs. In: Proceedings of the 40th Annual International Symposium on Computer Architecture, ISCA 2013, pp. 487–498. ACM, New York (2013)
37. Bakhoda, A., Yuan, G.L., Fung, W.W.L., Wong, H., Aamodt, T.M.: Analyzing CUDA Workloads Using a Detailed GPU Simulator. In: IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2009), pp. 163–174 (2009)
38. Top500: Top500 website, <http://www.top500.org/>
39. Green500: Green500 website, <http://www.green500.org/>
40. Kamil, S., Shalf, J., Strohmaier, E.: Power efficiency in high performance computing. In: IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, pp. 1–8 (2008)

41. Dongarra, J.: HPCG benchmarking,
<http://www.sandia.gov/~maherou/docs/HPCG-Benchmark.pdf>
42. Lee, V.W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A.D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R., Dubey, P.: Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. SIGARCH Comput. Archit. News 38, 451–460 (2010)
43. Dennard, R., Gaenslen, F., Yu, H.N., Leo Rideovt, V., Bassous, E., Leblanc, A.R.: Design of ion-implanted MOSFET's with very small physical dimensions. IEEE Solid-State Circuits Society Newsletter 12, 38–50 (2007)
44. Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K., Burger, D.: Dark silicon and the end of multicore scaling. SIGARCH Comput. Archit. News 39, 365–376 (2011)
45. Keckler, S., Dally, W., Khailany, B., Garland, M., Glasco, D.: GPUs and the Future of Parallel Computing. IEEE Micro 31, 7–17 (2011)
46. Wulf, W.A., McKee, S.A.: Hitting the memory wall: implications of the obvious. SIGARCH Comput. Archit. News 23, 20–24 (1995)
47. Patterson, P.D.: Latency lags bandwidth. In: Proceedings of the 2005 International Conference on Computer Design, ICCD 2005, pp. 3–6. IEEE Computer Society, Washington, DC (2005)
48. Cook, S.: CUDA Programming: A Developer's Guide to Parallel Computing with GPUs, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco (2012)
49. Hennessy, J.L., Patterson, D.A.: Computer Architecture, Fourth Edition: A Quantitative Approach. Morgan Kaufmann Publishers Inc., San Francisco (2006)

On the Performance Portability of Structured Grid Codes on Many-Core Computer Architectures

Simon McIntosh-Smith, Michael Boulton, Dan Curran, and James Price

Department of Computer Science, University of Bristol,
Woodland Road, Clifton, Bristol, BS8 1UB, UK
<http://www.cs.bris.ac.uk/home/simonsm/>

Abstract. With the advent of many-core computer architectures such as GPGPUs from NVIDIA and AMD, and more recently Intel’s Xeon Phi, ensuring performance portability of HPC codes is potentially becoming more complex. In this work we have focused on one important application area — structured grid codes — and investigated techniques for ensuring performance portability across a diverse range of different, high-end many-core architectures. We chose three codes to investigate: a 3D lattice Boltzmann code (D3Q19 BGK), the CloverLeaf hydrodynamics mini application from Sandia’s Manteko benchmark suite, and ROTORSIM, a production-quality structured grid, multiblock, compressible finite-volume CFD code. We have developed OpenCL versions of these codes in order to provide cross-platform functional portability, and compared the performance of the OpenCL versions of these structured grid codes to optimized versions on each platform, including hybrid OpenMP/MPI/AVX versions on CPUs and Xeon Phi, and CUDA versions on NVIDIA GPUs. Our results show that, contrary to conventional wisdom, using OpenCL it is possible to achieve a high degree of performance portability, at least for structured grid applications, using a set of straightforward techniques. The performance portable code in OpenCL is also highly competitive with the best performance using the native parallel programming models on each platform.

Keywords: Many-core, heterogeneous, GPU, Xeon Phi, structured grid, multi-grid multi-block, lattice Boltzmann.

1 Introduction

For the last ten years, Multi-core and Many-core computer architectures have become the main approach that processor vendors have taken in order to harness the ever greater numbers of transistors afforded by Moore’s Law [1]. At the time of writing, mainstream HPC processors containing 12, 16, and in the case of IBM’s Blue Gene/Q CPU, 18 cores, are not uncommon. Beyond multi-core CPUs, many-core processors, such as Intel’s Xeon Phi, NVIDIA’s Tesla GPUs and AMD’s FirePro GPUs, harness even greater numbers of cores with wide

vector units, delivering total hardware parallelism of the order of 10^3 ALUs in a single device. Hardware releases from the last few years, projected trends in silicon technology and vendor roadmaps together suggest that this trend of increasing numbers of cores with wide data parallelism per core is set to continue.

Memory hierarchies are also becoming increasingly complex. As memory latencies are fundamentally difficult to improve, being largely defined by the speed of light and the distances that data must travel, memory subsystems are instead adding more levels of hierarchy and greater parallelism (number of memory banks and/or controllers) to deliver greater address and data bandwidth.

This increasing complexity in both processor parallelism and memory hierarchy creates a major challenge for software developers: *performance portability*. Performance portability has never been free — for example, observe the evolution of the Basic Linear Algebra Subprograms (BLAS) from simple vector-vector operations (Level 1 BLAS) up to more sophisticated blocked matrix-matrix algorithms (Level 3 BLAS) in response to on-chip caches and increasingly hierarchical memories [2].

The potential for portable many-core software first became possible in 2008 with the arrival of the first open standard for heterogeneous many-core programming: OpenCL [3]. OpenCL was developed from the ground-up to support many-core style programs on multiple, heterogeneous devices. OpenCL thus solved the functional portability problem; programs written in OpenCL should run correctly on *any* OpenCL compatible device, from multi-core CPUs with SIMD instruction sets to many-core GPUs. But what OpenCL could *not* do, was guarantee *performance* portability out of the box for every application. Of course providing completely transparent performance portability for all OpenCL applications across all compatible hardware platforms was always going to be an unrealistic expectation. Indeed even CUDA developers have noted that their code often has to be tuned for different generations of NVIDIA hardware, even though their code is a single source with strong support from NVIDIA, and with hardware also coming from a single vendor. As an example, the Amber molecular dynamics package [4] includes a mature CUDA port [5, 6], but in order to achieve good performance across a range of different NVIDIA GPUs, four different codepaths were required, each with a different method of using single, double, integer and mixed precision arithmetic [7].

In response to the performance portability challenge, a number of different approaches emerged. Auto-tuning is one of the most widely explored methods for addressing performance portability [8, 9]. Auto-tuning employs techniques such as *meta programming*, where certain configurable characteristics of a target code can be altered automatically or semi-automatically as part of a platform-dependent optimization process. An auto-tuner modifies features such as degree of parallelism, block sizes, vector widths, data layout and even algorithm selection, in order to configure the code for optimal performance on a given target hardware platform.

The current conventional wisdom is that, while functional portability across many-core architectures is now possible with OpenCL, performance portability is

still an unsolved problem. Much research is now underway to try and solve this problem. Yet in some recent work in the HPC research group at the University of Bristol, the authors discovered an exciting counter example. The BUDE molecular docking code [10, 11] that we had been developing in OpenCL had been optimized for a specific NVIDIA GPU, a GTX 680. On this specific device we achieved a sustained single-precision performance of 1.43 TFLOP/s, representing a sustained performance across the whole application of 46% of the device’s peak. After optimizing the BUDE OpenCL code for this specific hardware platform, we then benchmarked exactly the same BUDE OpenCL code on a wide range of other platforms. The only changes allowed were simple, run-time choices for parameters such as the OpenCL work-group sizes; the source code itself remained identical across all tested platforms. We were surprised by the results: the BUDE OpenCL code sustained an average of 40% of peak performance across eight different hardware platforms, including consumer and HPC GPUs from both NVIDIA and AMD, and Intel Xeon and Xeon Phi. This result prompted the authors to ask the question: “If one HPC code shows good natural performance portability across many different hardware platforms, do any of the others?”. That is the question we have set out to start to answer in this paper. To focus our efforts, we have chosen to concentrate on *structured grid* codes [12].

1.1 Contributions

The specific contributions of the work presented in this paper include:

- We have developed optimized OpenCL versions of three non-trivial structured grid codes: an in-house D3Q19-BGK lattice Boltzmann code; ROTORSIM, a multiblock, multigrid, compressible finite-volume CFD code; and CloverLeaf, a hydrodynamics mini application that now forms part of Sandia’s Manteko benchmark suite.
- We present a rigorous study of the performance portability of these structured grid codes across a diverse range of contemporary multi-core and many-core parallel architectures, including an analysis of the fractions of peak floating-point and memory bandwidth the codes are achieving on each target device.
- We give an evaluation of OpenCL as a performance portability infrastructure for HPC application development. This includes the comparison of OpenCL performance to any native parallel programming environments for each target hardware platform, such as CUDA on NVIDIA GPUs or hybrid MPI and OpenMP on Intel CPUs and Xeon Phi.

2 Structured Grid Codes

Structured grids are one of the main application kernels, or *seven dwarfs* identified by Phil Colella in 2004 [13] before being popularized in the seminal ‘View from Berkeley’ paper from 2006 [12]. Structured grid codes are used to solve

many different kinds of numerical systems, for example solving the Navier Stokes equations in order to model fluid flows. Structured grids operate on a regular grid of data, with points on the grid being conceptually updated together. They typically have a high spatial locality, and updates to the grid points may be in place or between two versions of the grid. Structured grid codes are known to be highly scalable, and are highly amenable to spatial decomposition techniques to exploit massively parallel machines.

Many important HPC codes employ structured grid techniques: they form the basis of numerous lattice Boltzmann codes, Computational Fluid Dynamics (CFD) codes and Magneto Hydrodynamics codes, amongst many others. It is this combination of wide-spread use and known parallel scaling that lead us to choose structured grids as the subject for this initial investigation. We shall now introduce the three structured grid codes chosen for this study.

2.1 D3Q19-BGK Lattice Boltzmann

The lattice Boltzmann Method (LBM) has been gaining significant popularity in HPC as a versatile approach to solving incompressible flows based on a simplified gas-kinetic description of the Boltzmann equation [14–16]. LBM implementations utilize a uniform (structured) grid of lattice cells, which are updated based on nearest neighbor information at every time step. The dimensionality, number of nearest neighbors involved in cell updates, and relaxation term, are usually encoded into an LBM’s name. For example, D3Q19-BGK means a three dimensional lattice with 19 of the 27 possible nearest neighbors involved in each cell update (the 8 corners of the cube are ignored), and employing the Bhatnagar-Gross-Krook relaxation term. D3Q19-BGK is one of the most popular forms of LBM in use today [17].

For the purposes of this study, we have developed, from scratch, a single precision D3Q19-BGK LBM code, and provided optimized versions in a number of parallel programming languages in order to assess LBM’s performance portability across a range of target hardware platforms. Our LBM code supports OpenCL, CUDA, OpenMP, and on the CPU, AVX SIMD instructions. Performance tests show that our LBM code is competitive with the fastest reported in the literature; the best results we have found to date were reported by Mawson and Revell, who gave a D3Q19-BGK performance in CUDA on an NVIDIA K20 of 982 MLUPS (Millions of Lattice node Updates Per Second). This performance result was for single precision data on a 128^3 grid [18]. Januszewski and Kostur also recently published a study on the performance of their Sailfish LBM code, and reported a maximum performance of $\sim 1,000$ MLUPS for an NVIDIA K20 on a 256^3 single precision D3Q19-BGK case [19]. On the same device our D3Q19-BGK LBM code, written in OpenCL, achieved 1,110 MLUPS, a 13% improvement over the Mawson-Revell result, and a 10% improvement over the Januszewski-Kostur result. Thus we can be confident that our D3Q19-BGK LBM code is representative of the fastest LBM codes currently in use.

As with many structured grid applications, well optimized LBM codes are known to be memory bandwidth limited, and so if performance portability is

achievable for our LBM code, it will show the potential for performance portability for similar classes of problem.

2.2 ROTORSIM

Developed by Prof. Christian Allen at the University of Bristol, ROTORSIM is a multiblock, compressible finite-volume fluid simulation tool [20, 21]. It adopts an upwind, third-order accurate spatial stencil, with an explicit time integration scheme for steady flows and implicit dual-time approach for time-accurate calculations. Both schemes use efficient multigrid convergence acceleration. Coupled fluid-structural computations, aerodynamic shape optimization, data morphing and surrogate modelling have all been integrated within the code. ROTORSIM had already been parallelised using MPI, and showed excellent scaling [22]. The ROTORSIM CFD code and integrated technologies have been applied to fixed-wing and rotary-wing flows; for example, very challenging forward-flight rotor cases have been simulated using high quality mesh deformation [23], and the code was used to perform the first unsteady simulation of hovering rotor wakes [22]. Aeroelastic deformations and flutter boundaries have also been simulated [23], and aerodynamic optimisations performed for fixed- and rotary-wing cases [24]. [24] is believed to be the first free-form shape optimisation of a hovering rotor blade.

The authors have been contributing to a program of re-engineering ROTORSIM's 91,000 lines of Fortran into C in order to support long-term code development. At the same time, the authors have developed an OpenCL version of ROTORSIM, and it is this new code which has been used in this study. These are the first performance results published for the OpenCL version of ROTORSIM, which is available to license from Prof. Chris Allen at the University of Bristol.

2.3 CloverLeaf

For our third structured grid code, we turned to the CloverLeaf Lagrangian-Eulerian hydrodynamics code [25], one mini-app that forms part of the Manteko benchmark suite being developed by Sandia National Laboratories [26]. The code for CloverLeaf can be downloaded, along with other mini-apps, from the Manteko website [27].

The CloverLeaf mini-application solves the compressible Euler equations, a system of equations describing the conservation of energy, mass, and momentum in a system. These equations are solved on a Cartesian grid in two dimensions (development of a 3D version of CloverLeaf is currently underway). CloverLeaf solves the equations with second-order accuracy, using an explicit finite-volume method.

This simple hydrodynamics scheme is representative of that found in other hydrodynamics codes, but it has been developed in such a way that all CloverLeaf's numerical computation is carried out in a number of small kernel functions, making long, complex loops containing many subroutine calls unnecessary.

This approach makes profiling the code and analysing the results more straightforward, an important goal for a synthetically created benchmark.

CloverLeaf has been ported to a number of parallel programming languages, including OpenMP, MPI, OpenACC and Co-Array Fortran. The authors of this paper contributed to the development of the OpenCL and CUDA versions of CloverLeaf. Mallinson et al. recently showed that the OpenCL version of CloverLeaf performed almost identically to a native CUDA version on NVIDIA GPUs [28].

3 Experimental Platforms

For the purposes of this study we wished to analyse the performance of structured grid codes across a wide variety of many-core computer architectures. To this end, we chose a range of different CPU, GPU and accelerators from the three main many-core HPC processor vendors.

From NVIDIA, we used two HPC-optimized Tesla GPUs: a ‘Kepler’ architecture K20c and a ‘Fermi’ architecture Tesla M2090. To widen the range of NVIDIA devices in the study, we also selected two high-end GPUs from NVIDIA’s consumer range, including a GTX 780 Ti and a GTX 680. From AMD we chose their highest-end professional GPU, the S10000, based on their Tahiti architecture. We also chose two high-end AMD consumer GPUs, the Radeon R9 290X and the Radeon HD 7970. From Intel, we chose a 61 core Xeon Phi SE10P, and for comparison, we also used a dual socket, 16 cores total, Intel Xeon E5-2687W ‘Sandy Bridge’ CPU system. This model of x86 CPU includes support for the 256-bit wide AVX SIMD instruction set. In total we had nine devices in the study. The main characteristics of all these devices are listed alphabetically in Table 1.

These target devices were installed across three different test systems, as detailed in Table 2. We used GCC 4.8.2 to compile the host codes for the GPU devices, and Intel’s ICC 13.1 for the C and Fortran OpenMP tests on the Xeon Phi and Xeon CPU devices. The compiler flag “-fast” was used for ICC, which

Table 1. Specifications of target devices used for testing

Platform	Clock (GHz)	RAM (GB)	Memory B/W (GB/s)	S.P. TFLOP/s	D.P. TFLOP/s	TDP (W)
AMD FirePro S10000	0.825	6	480	5.91	1.48	375
AMD Radeon HD 7970	0.925	3	264	3.78	0.95	230
AMD Radeon R9 290X	1.000	4	320	5.63	0.70	250
Intel Xeon E5-2687W (x2)	3.100	32	102	0.79	0.40	300
Intel Xeon Phi SE10P	1.100	8	320	2.15	1.07	300
NVIDIA GTX 780 Ti	0.928	3	336	5.05	0.21	250
NVIDIA GTX 680	1.006	2	192	3.00	0.13	195
NVIDIA Tesla K20	0.706	6	208	3.52	1.17	225
NVIDIA Tesla M2090	0.650	6	177	1.33	0.66	225

Table 2. Benchmark system specifications

	Test system 1	Test System 2	Test System 3
Device(s)	AMD and NVIDIA GPUs except M2090	Xeon Phi SE10P, Xeon E5-2687W	M2090 GPU
CPU Model	Intel Core i5-3550 (x1)	Intel Xeon E5-2687W (x2)	Intel Xeon E5649 (x2)
CPU Specs	4 cores (x1), 3.3 GHz	8 cores (x2), 3.1 GHz	6 cores (x2), 2.53 GHz
Memory	16 GB	32 GB	48 GB
OS	Ubuntu 12.04	Red Hat Enterprise Linux 6.3	Red Hat Enterprise Linux 6.4
Driver/SDK	CUDA 5.5.22 / 331.49, AMD APP SDK 2.9.1.1.1 / 14.3beta	Intel SDK for OpenCL Applications 2013 / 3.2.1.16712	CUDA 5.5.22 / 319.76

turns on a range of aggressive optimisations, including -O3, automatic vectorisation, and use of prefetching.

4 Methodology

To accurately assess the performance portability of the three structured grid codes in the study, the following steps were taken.

First, for each structure grid code, a fixed sized problem was chosen which was both realistic (i.e. not larger or smaller than is interesting for the application) and large enough for each device to display performance near the asymptote that one would expect. This same fixed problem size and input data was then used for every run on every device, enabling fair comparisons.

Next we ensured that exactly the same code was executed on each device. In the case of the OpenCL versions of each code, this means that all the code, including both the kernel code and the host code was identical across devices. The same was true for the CUDA runs on the NVIDIA GPUs, and the OpenMP/MPI and OpenMP+SIMD runs on the Xeon and Xeon Phi.

For the OpenCL and CUDA runs, the work-group or thread-block sizes were either fixed at the same value for all devices, or left up to the run-time to decide. The D3Q19-BGK and CloverLeaf codes took the first approach, while the ROTORSIM code took the latter. ROTORSIM relies on the OpenCL run-time to choose the work-group or thread-block sizes because it has many different kernels and these are compiled for different sizes of data block. Thus any one choice for a work-group or thread-block is likely to be suboptimal across the range of kernels and data to be processed. By leaving the choice up to the run-time at the point of kernel execution, more optimal per kernel work-group/thread-block size choices can be made by the run-time.

Where OpenCL and CUDA versions of a code were used, we took every effort to ensure that these codes were as similar as possible. In most cases if one

of the kernels already existed in either OpenCL or CUDA, then this version of the code was directly used to create the complementary version. Also, the OpenCL and CUDA computational kernels were kept as similar as possible to the original C code, and thus were almost identical to the hybrid OpenMP/MPI and OpenMP+SIMD versions, again enabling direct, fair comparisons between different devices. Thus all code versions received a similar degree of optimisation.

For the SIMD versions of the LBM code for the Intel Xeon and Xeon Phi devices, the OpenMP code was extended manually using explicit AVX vector intrinsics for the Xeon CPU. For the Xeon Phi we relied on automatic compiler vectorisation, and confirmed that this had been successfully carried out by checking the appropriate compiler report. As structured grid codes are very regular in their structure, in many cases the main loops of the scalar OpenMP code had been completely vectorized by ICC for both Xeon and Xeon Phi. This shows as only a small difference between the OpenMP and OpenMP+SIMD times in some of the results.

Finally, the same versions of software were used across all three test machines, particularly device SDKs, drivers and compilers, and each test was performed five times and a mean average taken. All of these steps in combination meant that like-for-like comparisons could be made between the test devices.

5 Results

5.1 D3Q19-BGK Lattice Boltzmann

The D3Q19-BGK code developed for this study parallelized the computation by allocating one grid point to each OpenCL work-item or CUDA thread. The OpenMP version of the C code used OpenMP pragmas to annotate the main computational loops, leaving it to the OpenMP runtime to decide how best to distribute the loop iterations over the available cores. For the Xeon CPU and Xeon Phi we relied on the Intel compiler to vectorise the code for us, and verified that it had successfully done so by examining the appropriate compiler reports. The code was implemented in a fairly straightforward manner, translating the basic algorithm directly but efficiently into the various parallel programming languages under test, with the one optimisation to use a tiled approach. This interleaved the 19 input arrays holding the 19 speeds so that all the data accessed together, for example by an OpenCL work-group, ended up as one contiguous block of data, rather than being distant from one another in memory. This approach was suggested by Intel's Xeon Phi OpenCL compiler team, and resulted in greater performance on every device, as it helped to reduce the number of simultaneously open memory pages, and thus helped reduce cache and TLB misses.

Lattice Boltzmann performance is often quoted in units of Millions of Lattice Updates Per Second, or MLUPS. For a given MLUP rate, p , it is possible to calculate the sustained bandwidth, b , that an LBM run achieved. For a single-precision D3Q19 this formula is:

$$b = \frac{p \times 10^6 \times 19 \times 2 \times 4}{1024^3}. \quad (1)$$

The 10^6 term merely translates the MLUP rate into LUPS; the factor of 19 comes from having 19 speeds for each cell point; the factor of 2 represents a read and a write for each item of data; the 4 is the number of bytes in the single precision data we were using for these experiments; and finally, the 1024^3 scales the measured sustained bandwidth from bytes per second to Gigabytes per second.

For the D3Q19-BGK test run for this study, we set the following parameters:

- The single precision grid was fixed at 128^3 in size for all runs on all devices. This resulted in a total of $\sim 2m$ grid points, which with 19 speeds per grid point resulted in a total working memory space of 304 MBytes.
- The OpenCL three dimensional work-group size was fixed at (128,1,1) for all OpenCL runs on all devices.
- The CUDA thread grouping was arranged in exactly the same way as the OpenCL execution, with a `blocksize` of (128,1,1) for all NVIDIA GPUs.
- The AMD S10000 results are shown for two different scenarios: using one or both of the two GPUs on the device.

The performance results for our D3Q19-BGK LB are shown in Figure 1a. A first observation is the relative performance equality between the OpenCL and CUDA versions of the code, with an average difference of 8.1%. Perhaps surprisingly, the OpenCL code was faster in two of the three cases, and on NVIDIA’s most recent GPUs, the Tesla K20c and the GTX 780 Ti, the differences were 15.1% and 15.3% in favor of OpenCL, respectively.

Also of interest is the relative performance of OpenCL and OpenMP on the Xeon and Xeon Phi. On the Xeon CPU the OpenCL code is 86% faster than compiler-vectorised OpenMP code on the same device. On the Xeon Phi situation is reversed, with the compiler-vectorised OpenMP code exceeding the performance of the OpenCL code by 39%. Inspection of the vectorisation reports

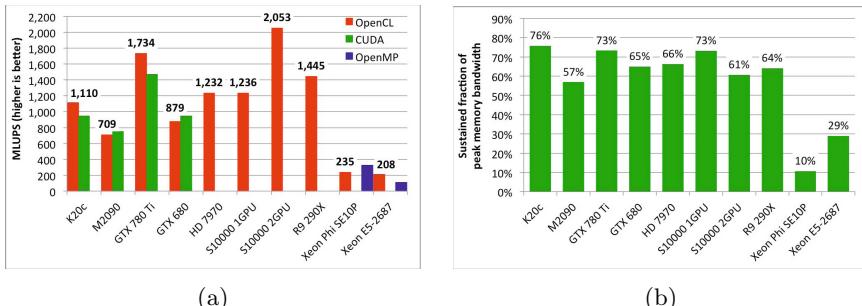


Fig. 1. D3Q19-BGK performance. Figure 1a shows MLUPS on the vertical axis, while Figure 1b shows the fraction of peak memory bandwidth sustained during the benchmark runs (higher is better in both graphs).

from Intel’s compiler confirmed that ICC was successfully vectorising the main loop of the OpenMP LBM code on both Intel target platforms. The competitiveness of the OpenCL results also suggests that Intel’s OpenCL implementation appears to be effective in packing OpenCL work-items into SIMD lanes, both on the Xeon CPU and the Xeon Phi.

While Figure 1a is interesting in terms of showing the relative performance of each of the devices, it does not in itself tell us anything about the performance portability of our D3Q19-BGK code. To achieve this, one can look at the fraction of peak performance each device achieves. If we see a similar high fraction of peak performance sustained across the target devices, then we would claim that our code was *performance portable*. Given than structured grid codes in general, and LBM codes in particular, tend to be bandwidth limited, Figure 1b shows the fraction of peak bandwidth each run sustains. These results are all based on running the identical OpenCL code on every device.

There are a number of important observations to make about these results. First of all, one might hope that with a well optimized structured grid code, one could achieve a high fraction of the peak memory bandwidth of each device. And indeed, across all the devices in the test, the average fraction of peak memory bandwidth that is sustained by our D3Q19-BGK LBM code is 57%. Seven of the nine devices exceeded half of their peak memory bandwidth for this application, with three devices, the K20c, GTX 780 Ti and S10000 sustaining over 70% of their peak memory bandwidth (where we count the S10000 as a single device). Two devices do not fair so well; the Xeon CPU achieves 29% of its peak memory bandwidth, a respectable figure given the relatively large size of the dataset and the strided nature of processing a 3D LBM data structure.

The Xeon Phi fairs least well in this test, only achieving 10% of its peak memory bandwidth. We do not yet fully understand the reasons behind the Xeon Phi’s low performance on this benchmark, but observe that there are at least two issues in play. The first issue is that the Xeon Phi’s OpenCL toolchain is still maturing. Our BUDE OpenCL molecular docking code has improved from sustaining 5% of peak to 32% of peak on a Xeon Phi over the last two releases of Intel’s Xeon Phi OpenCL SDK [10], and so we would expect that the Xeon Phi’s performance on structured grid codes will continue to improve as the toolchain matures. The second issue is that the Xeon Phi can, at best, only achieve 52% of its peak memory bandwidth (as indicated by a Stream Triad result of 165 GBytes/s vs. a peak memory bandwidth of 320 GBytes/s). We would therefore never expect to exceed this 52% for our lattice Boltzmann code, and a more realistic target for the Xeon Phi would be some fraction of this figure.

In terms of absolute performance, the fastest device in the test is the AMD S10000, where this device’s 480 GBytes/s of peak memory bandwidth delivers an MLUPS rate of 2,053 for our 128^3 D3Q19-BGK single precision test case. To the best of our knowledge, this is the highest reported rate for a single device to date, and the first time that a single device has broken the 2,000 MLUPS barrier for the 128^3 single precision test case. It could be argued that the S10000 is not strictly a single device, given the dual GPU chips on this accelerator. The

fastest single GPU device is the GTX 780 Ti, with an MLUPS rate of 1,734 for the same 128³ D3Q19-BGK single precision test case.

5.2 ROTORSIM

The ROTORSIM code again parallelizes by allocating grid points to work-items in OpenCL. No CUDA or OpenMP ports of this code exist, so we were limited to measuring the relative fraction of peak FLOP/s and memory bandwidth the OpenCL version of ROTORSIM achieved across our range of target devices.

For the ROTORSIM test run for this study, we set the following parameters:

- We set the work-group size to ‘NULL’ on every device. This lets the OpenCL run-time make the decision for the best work-group size, for reasons already explained in Section 4.
- The example data set was ‘cylinder’, a 256x128x32 3D grid of double precision data organised as one large block. Three multigrid levels were employed.
- The AMD S10000 results used just one of the two GPUs on the device, as at the time of writing, ROTORSIM was designed to run on a single device. As the dual GPUs on an S10000 are essentially independent, each with its own memory and memory bandwidth, we have simply halved the peak performance of the device for the purposes of reporting the results below.
- the Intel Xeon Phi SE10P was left out of these results as we encountered problems with its OpenCL drivers when trying to run the ROTORSIM benchmark.

ROTORSIM was already understood to be a bandwidth limited code, and thus we would expect the devices with the highest memory bandwidth to do well on this benchmark. However, in contrast to our single precision D3Q19-BGK LBM code, ROTORSIM operates on *double precision* data. Several of our target devices, specifically the consumer-oriented GPUs, have very high memory bandwidth, but relatively low double precision performance compared to their single precision capability. Thus the interaction between a device’s memory bandwidth and double precision floating point performance results in a more complex situation than for our D3Q19-BGK LBM.

In order to show the impact of each of these performance characteristics, we show ROTORSIM’s performance results in three different ways. In Figure 2a we show ROTORSIM’s absolute run-time performance, measured in ‘cycles per second’. A cycle in ROTORSIM consists of running the solver at three different levels of coarsening, with a total of three iterations each at the finest and middle levels of coarsening, and two iterations at the coarsest level. (For an in-depth discussion of algebraic multigrid methods, we refer the interested reader to [29]). Then, in Figure 2b we show the sustained fraction of peak memory bandwidth (top), and ROTORSIM performance relative to each device’s peak double precision floating point (bottom).

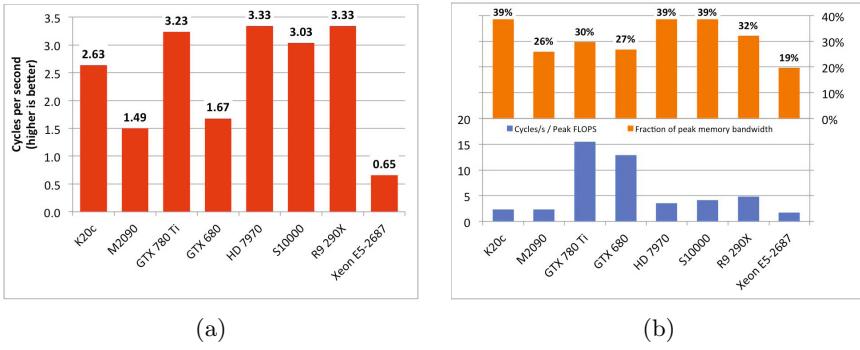


Fig. 2. ROTORSIM performance. Figure 2a shows performance in cycles per second. Figure 2b shows the sustained fraction of memory bandwidth on each device (top), and performance relative to each device’s peak double precision floating point capability (bottom).

The absolute performance in Figure 2a shows that, perhaps as expected, the devices with the highest memory bandwidth tend to deliver the highest performance for this application. However, these absolute performance results alone do not yield any insight into the relative performance portability of ROTORSIM across the devices.

In this regard Figure 2b is much more illuminating. The top graph, which shows the sustained fraction of peak memory bandwidth, reveals that once again, we see most of the devices in the test achieving a similar fraction of their peak memory bandwidth, averaging 31% across all the devices. This is compared to the 57% of the much more regular D3Q19-BGK LBM code described in the previous section. Given how much more complex the ROTORSIM code is relative to the D3Q19-BGK LBM, we were satisfied that on average each device is sustaining about 1/3rd of peak memory bandwidth across a complex, production-quality, multi-block multi-grid CFD code that runs dozens of different OpenCL kernels during its execution. The three devices that sustained the highest fraction of peak memory bandwidth, all at 39%, were the K20c, the HD7970 and the S10000. The slowest device in this group was the Xeon CPU, which still managed to sustain a respectable 19% of peak memory bandwidth when running the same OpenCL code, not too far behind the slowest GPU, the M2090 at 26%.

The sustained memory bandwidth results are relatively closely clustered. Three of the eight results (NVIDIA K20c, AMD HD 7970 and S10000) were almost identical. Of the three results which were most distant from the average (the M2090, GTX 780 Ti and GTX 680), much of this difference can be attributed to the relatively low double precision performance of the consumer-oriented GPUs. While the high memory bandwidth of the GTX 780 Ti and GTX 680 meant they performed very well on the single precision only LBM, their low double precision performance has had a significant impact on ROTORSIM performance. This can be seen in the lower graph in Figure 2b, where their performance relative to their peak double precision performance stands out

as high compared to the other devices. This result confirms that ROTORSIM is largely memory bound – even a GPU with relatively low double precision performance can perform well for ROTORSIM if it has enough memory bandwidth to overcome this deficiency.

Table 1 lists the relative single and double precision peak performances of the devices. The NVIDIA GTX 780 Ti has the second highest peak memory bandwidth in the table (336 GBytes/s) while its peak double precision performance is a mere 0.21 TFLOP/s, $1/24^{th}$ of its single precision performance. The GTX 680 also has a high peak memory bandwidth (192 GBytes/s) with a relatively low peak double precision performance of 0.13 TFLOP/s (also approximately $1/24^{th}$ of its single precision performance). These are by far the highest ratios of single precision to double precision peak performance, and this relatively low peak double precision capability has had a clear impact on ROTORSIM performance for these two devices.

To summarise the ROTORSIM results, again the achievable memory bandwidth dominates the delivered performance, and most of the target devices in the test achieve similar fractions of their peak memory bandwidth, with a 20% spread from the highest to lowest sustained memory bandwidth and an average of 31%. These results indicate that the OpenCL version of ROTORSIM is performing well, achieving a good fraction of peak memory bandwidth while also delivering reasonable performance portability. Where devices do not perform quite so well, this is largely due to significantly lower relative double precision floating point performance. However, it is interesting to observe that even though the NVIDIA GTX 780 Ti has the second lowest peak double precision performance of all the devices in the study, it still delivers close to the highest performance for ROTORSIM at 3.23 cycles per second, just behind the AMD HD 7970 and R9 290X, both of which deliver 3.33 cycles per second. As expected, floating point performance is much less important than memory bandwidth for ROTORSIM. If we consider just the set of four GPUs which have high double precision floating point (Nvidia K20c, AMD HD 7970, S10000 and R9 290X), the performance portability is very strong, with these devices achieving a sustained fraction of peak performance of between 32% and 39%.

5.3 CloverLeaf

For the CloverLeaf benchmark runs for this study, we set the following parameters:

- The double precision grid was fixed at 1920×3840 in size for all runs on all devices. This resulted in a total of $\sim 7.4\text{m}$ grid points, which with 25 values per grid point resulted in a total working memory space of ~ 1.5 GBytes.
- The OpenCL and CUDA parallelizations were performed in an identical manner, allocating one work-item or thread to each grid point, and with identical arrangements for work-group sizes and layouts. In the OpenCL case we used 2D work-groups of (128, 1) work-items, while in CUDA we used 1D blocks of 128 threads.

- The AMD S10000 results are shown for two scenarios: using one or both of the two GPUs on board. To use both GPUs on the S10000 we employed two MPI tasks on the host to drive the dual on-board GPUs as if they were independent devices. As the dual GPUs on an S10000 are essentially independent, each with its own memory and memory bandwidth, the results using both GPUs on the S10000 were almost exactly double the single GPU performance.
- For the OpenMP version of CloverLeaf, no explicit SIMD vectorization was performed. Instead we relied on the Intel compiler, ICC, to perform this vectorization for us. Upon examining the vectorization reports from the compiler, it was clear that ICC was indeed vectorizing all of the main loops for CloverLeaf for both the Xeon CPU and Xeon Phi.
- We employed hybrid OpenMP/MPI parallelization strategies to achieve the best performance on the Intel Xeon CPU and Xeon Phi. On the Xeon CPU, we allocated one MPI task per CPU (socket), and the one OpenMP thread per core on each CPU. On the Xeon E5-2687W dual CPU system, this resulted in two MPI tasks, each with eight OpenMP threads (sixteen OpenMP threads in total). On the Xeon Phi SE10P, the arrangement was one MPI task per core, with four OpenMP threads per MPI task, resulting in 60 MPI tasks, each with four OpenMP threads (240 OpenMP threads in total).

CloverLeaf performance is reported in iterations per second, with higher numbers being better; see Figure 3a. Then, in Figure 3b we show the sustained memory bandwidth as a fraction of peak on each device (top), as well as CloverLeaf performance relative to each device’s peak double precision floating point (bottom).

The performance results have a lot in common with those reported for ROTORSIM in Section 5.2. Figure 3a shows that, once again, the devices with the highest memory bandwidth tend to deliver the highest absolute performance for CloverLeaf, with the AMD GPUs delivering consistently the highest

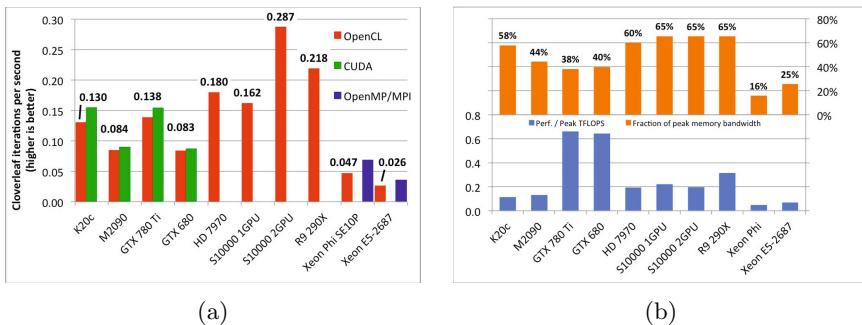


Fig. 3. CloverLeaf performance. Figure 3a shows performance in iterations per second. Figure 3b shows the sustained fraction of peak memory bandwidth (top), and performance relative to peak double precision floating point (bottom).

performance overall. The AMD S10000 delivers by far the highest performance in this test when using both of the on-board GPUs in the device. The S10000's 480 GBytes/s of peak memory bandwidth translated into a significant performance advantage, delivering 312.6 GBytes/s sustained for CloverLeaf, enough to achieve a 24% performance lead over the second fastest device, the R9 290X. Those devices with high memory bandwidth but very low double precision performance, such as the NVIDIA GTX 780 Ti and GTX 680, fared less well. Some of CloverLeaf's kernels are compute bound and require double precision floating point, which is why we see reduced performance on those devices which have lower double precision performance. The Xeon CPU performed very well relative to its peak memory bandwidth, while the Xeon Phi found it difficult to deliver good performance for this benchmark.

There are a number of other interesting observations to make regarding Figure 3a. Firstly, the OpenCL and CUDA versions of CloverLeaf perform very similarly, with usually a small performance advantage to the CUDA version of the code (averaging 9%). Secondly, on the Xeon CPU, the hybrid OpenMP/MPI version of CloverLeaf has a performance advantage over the OpenCL version of 28%, with a similar sized difference between the two versions on the Xeon Phi of 32%.

The relatively low performance of the Xeon Phi warranted further investigation. Using OpenCL's built-in profiling capability via OpenCL events, and making use of Extrae and Paraver [30] to gather and visualize the events, we discovered that two specific kernels performed much slower than expected compared to the other platforms. These kernels were for performing the left and right halo updates, which require strided memory accesses. This one issue reduced the OpenCL performance on the Xeon Phi relative to the OpenMP/MPI hybrid code, and indeed, the equivalent code sequence in the OpenMP/MPI hybrid performed as expected on the Xeon Phi. Therefore we believe that this issue is related to sub-optimal code generation in the Xeon Phi OpenCL toolchain, rather than related to the Phi's architecture. Further investigation is required to get to the bottom of this issue.

In order to consider the sustained fraction of peak memory bandwidth and performance relative to each device's peak FLOP/s, Figure 3b shows a picture consistent with the ROTORSIM and D3Q19-BGK LBM results. Five of the ten target devices, including the four fastest results, show a very similar sustained fraction of peak memory bandwidth. Notably the AMD GPUs tended to sustain the highest memory bandwidth and thus performance for CloverLeaf, with NVIDIA's K20c and GTX 780 Ti also delivering strong performance. The NVIDIA consumer GPUs, the GTX 780 Ti and GTX 680, both achieve a slightly lower fraction of their peak memory bandwidth, but the lower graph in Figure 3b shows that, relative to their double precision floating point performance, these two devices significantly outperformed the others in the test. Clearly the low relative double precision performance of these two devices had an impact on their absolute performance for CloverLeaf, and yet both delivered good overall performance as a result of their high memory bandwidth. The Xeon Phi

delivered the lowest fraction of its peak memory bandwidth. Much of this can be attributed to the two slow halo update kernels described earlier, and also to the lower fraction of peak memory bandwidth that is achievable in general on the Xeon Phi, as previously described in the earlier section on ROTORSIM.

5.4 The Impact of Work-Group Sizes

One of the innovations in this study was to execute exactly the same code configured in exactly the same way across all devices. This approach is contrary to conventional wisdom, which would suggest that each device should receive its own tuning and code optimization. One specific choice we made was to always select work-groups of size 128 (or the CUDA equivalent). Work-group size is often noted as one of the main tuning parameters for OpenCL and CUDA [9], and so this choice may have had a large impact on the performance we have observed.

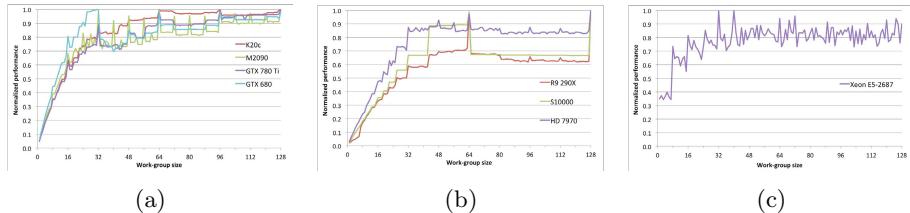


Fig. 4. Impact of work-group size on D3Q19-BGK LBM performance on the NVIDIA (Figure 4a), AMD (Figure 4b) and Intel CPU (Figure 4c) target devices. Normalized against best performance per device.

In order to evaluate the effects of our constant work-group size of 128, we investigated the impact on performance of varying the work-group size for the D3Q19-BGK LBM benchmark. As the dataset itself was 3D, we used a 3D work-group arrangement, while only varying the first (x) dimension of the work-group size from 1 up to 128. The other two dimensions of the 3D work-group size (y and z) were kept constant at 1 as we found that any other combination of 3D work-group sizes resulted in performance no better than this arrangement. As the parallelization technique was to allocate cells to work-items, and the test grid was 128^3 , 128 was the largest size in any one dimension that we could evaluate. Figure 4a, Figure 4b and Figure 4c show normalized performance for the NVIDIA, AMD and Intel target devices, respectively. Performance is normalized per device.

The GPU devices from AMD and NVIDIA show similar characteristics. Work-group sizes that are multiples of 16 are peaks, with the highest peaks at 32, 64, 96 and 128 for NVIDIA, and at 64 and 128 for AMD. These sizes correspond to the SIMD widths of the architectures, where NVIDIA has a 32-wide natural data

parallelism, or ‘warp’, and AMD has a corresponding 64-wide natural parallelism, or ‘wavefront’. In both cases, a 3D work-group size of (128,1,1) was optimal.

The Intel Xeon CPU shows slightly noisier work-group size behavior, but again with peaks at multiples of 8, and its highest performance peaks at 32 and 40. Generally though the Xeon CPU’s troughs are much higher, meaning that there are fewer bad choices of work-group sizes. Any multiple of 8 larger than 32 works well. A work-group size of 128 was not the optimal for the Xeon CPU, although this size did perform at 90% of the optimum.

In summary, for the D3Q19-BGK LBM code, a 3D work-group size of (128,1,1) proved optimal in almost all cases. Work-group sizes which are multiples of 64 in the major dimension are generally peaks in performance, with 128 often being the optimum, especially on many-core devices. Optimal work-group sizes for CPU devices can tend to be smaller, but a work-group size of 128 also results in high performance on the CPU.

6 Discussion

We initially found the results of this study quite surprising. Although we had already seen one example of an OpenCL code which had shown a high degree of performance portability without extensive per platform optimization or auto-tuning, we had expected that the more complex nature of the structured grid codes we were investigating would negate any natural performance portability of the underlying algorithms. We also had some expectation that the memory bandwidth limited nature of structured grid codes may have also limited their performance portability.

The evidence was quite contrary to our expectations. The D3Q19-BGK LBM code, while the simplest of the three benchmarks in the study, was still a non-trivial application, and is now amongst the highest performing LBM codes reported in the literature. As well as uncovering a new D3Q19-BGK performance champion (NVIDIA’s GTX 780 Ti, with 1,705 MLUPS on the 128^3 single precision test case), the code exhibited a very high degree of natural performance portability, with almost every device sustaining between 50 and 70% of its peak memory bandwidth. Recall this is when running identical OpenCL code configured in an identical manner on all the devices (3D work-group size of (128,1,1)).

ROTORSIM is a production quality CFD code that has been ported to OpenCL. While it is much more complex than the D3Q19-BGK LBM benchmark, similar results were observed, although ROTORSIM’s requirement for double precision floating point had a significant and detrimental impact on the performance of the NVIDIA consumer GPUs. Even so, the performance each device delivered relative to its peak memory bandwidth was remarkably consistent, again demonstrating strong performance portability. These results were closely repeated with the CloverLeaf hydrodynamics mini-app.

We have thus shown that, contrary to conventional wisdom, it *is* possible to develop very high performance codes which *are* performance portable across a diverse range of modern many-core architectures, *at least in the case of structured*

grid applications. This result also demonstrates that, even though one might expect that codes which are memory bandwidth bound would be more difficult than compute bound codes to make performance portable, it can be done.

We believe that part of the reason why conventional wisdom is wrong in this regard is that most of the studies into performance portability were performed early on, not long after many-core device architectures and the accompanying parallel programming languages of OpenCL and CUDA first emerged. At this time the software toolchains and drivers were immature, while the architectures themselves were also more difficult to generate efficient code for, as they still retained many features from their graphics processor heritage. In the University of Bristol HPC group we have been using OpenCL since 2009 to port HPC applications, and in our own experience we have seen significant improvement in the performance of OpenCL code on many-core devices from all the major HPC processor vendors. Thus the data on which much of the many-core performance portability conventional wisdom is based is now out of date. We hope that this paper will go some way to address this issue.

Finally, we have also shown that excellent performance can be achieved *without* the need for complicated auto-tuning techniques or device-specific code optimisation. All the results presented in this paper were running identical code parameterised in an identical manner wherever possible. We demonstrated that work-group sizes with major dimensions that were multiples of 64 generally performed well, and that a static choice of 128 was often optimal.

7 Related Work

In this work we have focused on the performance portability of structured grid codes and shown that, by adopting relatively straightforward techniques, it is possible to achieve high performance per device without having to make the code unduly complex. Partly this work is timely because OpenCL support is now widespread amongst all the main HPC processor vendors, and OpenCL tools and drivers are now mature. OpenCL is the only open, many-core programming standard widely supported across many-core processors (GPUs, CPUs and Xeon Phi), and thus it is currently the only parallel programming language in which a single source code can be written and then executed across almost all HPC many-core devices. OpenCL adoption in HPC is starting to increase rapidly; according to Google Scholar, the number of papers published in 2013 mentioning OpenCL grew by 31% relative to 2012, to 3,580 (by comparison the number of papers published mentioning CUDA, while much higher, showed its first annual decline in 2013, down 6% to 9,980). This increase in the adoption of OpenCL in HPC is a reflection of the relatively recent maturing of OpenCL SDKs and drivers, and the recent strong promotion of OpenCL for HPC many-core programming by Intel and AMD. Others have noted that there is still a relative lack of research exploring the potential for performance portability enabled by OpenCL [28].

Early work exploring performance portability relied on immature software tools and drivers, and tended to conclude that it was challenging to deliver

[31–34]. But more recent work has started to observe real performance portability as a possibility enabled by OpenCL. For example, work by Du et al in 2012 described the adoption of OpenCL for a performance and platform portable version of the MAGMA BLAS library [35, 36]. In a seminal performance portability paper from ISC 2013, Zhang et al demonstrated that performance portability using OpenCL was becoming feasible [9]. Importantly, their study included a diverse range of target devices, including GPUs, CPUs and an APU (integrated CPU and GPU in a single device).

Looking more specifically at structured grid applications, much work already exists regarding the optimization of lattice Boltzmann codes for many-core architectures. Stand out work in this area includes Januszewski and Kostur’s ‘Sailfish’, an open source fluid simulation package implementing the lattice Boltzmann method on GPUs using both CUDA and OpenCL [19]; Habich et al’s study into D3Q19 performance on NVIDIA GPUs [17, 37]; Gray and Stratford’s ‘Ludwig’ LBM optimized for GPUs [38, 39]; Xiong et al’s massively scalable LBM for GPUs [40]; and Geveler et al’s real-time CFD using LBM on a range of different architectures, including IBM’s Cell, multi-core CPUs and GPUs [41]. While most of this prior work focuses on optimizing LBM codes specifically for each architecture, ours is the first to explore the opposite approach, where we have tried to maximize performance of an identical LBM code across a diverse range of target devices. As stated previously, our D3Q19-BGK LBM is as fast as any single device LBM quoted in the literature, and our performance of 1,705 MLUPS in single precision on an NVIDIA GTX 780 Ti is faster than any other quoted performance we have found to date.

Turning to ROTORSIM, other non-LBM, structured grid CFD codes have been ported to many-core architectures such as GPUs and Xeon Phi, often to great effect. Brandvick and Pullan’s work on porting a 3D Euler solver to GPUs was one of the first research projects of its kind [42]. In 2008 Elsen et al reported significant speedups achieved by their GPU port of the Navier-Stokes Stanford University Solver (NSSUS) [43]. In 2009 Cohen et al reported their efforts to port a double precision structured grid CFD code to CUDA [44]. Also in 2009 Göddeke et all explored the GPU acceleration of an unmodified parallel finite element NavierStokes solver [45], while Phillips et al described their GPU port of the multi-block CFD code MBFLO [46]. Much of this work has focused on optimizing non-LBM structured grid codes for specific devices. In contrast, our work in porting ROTORSIM to OpenCL has focused on delivering portable performance across a diverse range of target devices. It is also one of the first CFD ports to support an efficient implementation of multi-block multi-grid methods.

For our final structured grid benchmark code, CloverLeaf has already been the subject of some research into performance portability [25, 28]. CloverLeaf is now part of the Manteko mini-app benchmark suite from Sandia National Laboratory [26, 27, 47]. Our contribution was the CUDA version of the CloverLeaf code, to complement the OpenMP, MPI, OpenACC and OpenCL versions that already existed. We also helped to re-engineer the OpenCL code and have

improved the overall performance portability of the benchmark. For an overview of related work, the interested reader is referred to [48].

8 Conclusions

In this work we have shown that it is possible for structured grid codes written in OpenCL to exhibit good performance portability across a diverse range of many-core processors, and even to a lesser extent across multi-core CPUs. We have also shown that this is achievable with relatively straightforward techniques, and in particular the performance portability we demonstrated did *not* rely on any auto-tuning or device-specific code optimizations. Instead we ran identical OpenCL code for each of our structured grid codes on all target devices, even choosing identical work-group sizes wherever possible.

We have also shown that structured grid codes are still largely memory bandwidth bound, with an important implication that many-core processors which have high memory bandwidth but low double precision floating point performance may still deliver high performance for double precision codes.

While this study has focused on just one particular member of the seven dwarfs family, the codes we have considered are non-trivial and thus we believe that our results should be relevant to many other structured grid applications. We thus hope that this paper will start to address the HPC performance portability ‘myth’, that software developed for many-core processors has no alternative but to be tuned for specific devices, sometimes with proprietary programming languages.

We hope that the HPC community will take this result for structured grid codes and explore the performance portability of other algorithmic classes from the seven dwarfs. It is quite likely that some dwarfs will prove to be more performance portable than others. We look forward to the undoubtedly surprising results that others will discover as we together attempt to find pragmatic solutions for many-core software performance portability.

Acknowledgments. The authors would like to thank Wayne Gaudin and Andy Herdman from the High Performance Computing group at AWE plc for their contributions to the section describing CloverLeaf. We would also like to thank Chris Allen for access to the ROTORSIM CFD code. This work has been supported by EPSRC grant EP/K038486/1 and the ASEArch CCP (EPSRC grant EP/J010553/1), and by financial support from AWE and the University of Bristol’s Intel Parallel Computing Center (IPCC). We would also like to thank AMD, Intel and NVIDIA for the donation of some of the equipment used in this study. We made use of the Centre for Innovation’s ‘Emerald’ GPU supercomputer in order to benchmark the M2090 GPUs. Finally, the University of Bristol’s Advanced Computing Research Centre, ACRC, provided access to some of the equipment for this paper, including the Blue Crystal supercomputer.

References

1. Moore, G.: Cramming more components onto integrated circuits. *Electronics Magazine*, 114–117 (April 1965)
2. Demmel, J., Dongarra, J., Parlett, B., Kahan, W., Gu, M., Bindel, D., Hida, Y., Li, X., Marques, O., Riedy, E.J., et al.: Prospectus for a dense linear algebra software library (April 2006)
3. Munshi, A. (ed.): The Khronos OpenCL Working Group: The OpenCL specification (2008)
4. Case, D., Darden, T., Cheatham III, T., Simmerling, C., Wang, J., Duke, R., Luo, R., Walker, R., Zhang, W., Merz, K., et al.: AMBER 2012. University of California, San Francisco (2012)
5. Götz, A.W., Williamson, M.J., Xu, D., Poole, D., Le Grand, S., Walker, R.C.: Routine microsecond molecular dynamics simulations with AMBER on GPUs. 1. Generalized Born. *Journal of Chemical Theory and Computation* 8(5), 1542–1555 (2012)
6. Salomon-Ferrer, R., Götz, A.W., Poole, D., Le Grand, S., Walker, R.C.: Routine microsecond molecular dynamics simulations with AMBER on GPUs. 2. Explicit Solvent Particle Mesh Ewald. *Journal of Chemical Theory and Computation* 9(9), 3878–3888 (2013)
7. Grand, S.L., Götz, A.W., Walker, R.C.: SPFP: Speed without compromise—a mixed precision model for GPU accelerated molecular dynamics simulations. *Computer Physics Communications* 184(2), 374–380 (2013)
8. Davidson, A., Owens, J.: Toward techniques for auto-tuning gpu algorithms. In: Jónasson, K. (ed.) PARA 2010, Part II. LNCS, vol. 7134, pp. 110–119. Springer, Heidelberg (2012)
9. Zhang, Y., Sinclair II, M., Chien, A.A.: Improving performance portability in OpenCL programs. In: Kunkel, J.M., Ludwig, T., Meuer, H.W. (eds.) ISC 2013. LNCS, vol. 7905, pp. 136–150. Springer, Heidelberg (2013)
10. McIntosh-Smith, S., Price, J., Sessions, R.B., Ibarra, A.A.: High performance in silico virtual drug screening on many-core processors. *International Journal of High Performance Computing Applications (IJHPCA)* (April 2014)
11. McIntosh-Smith, S., Sessions, R.B.: An accelerated, computer assisted molecular modeling method for drug design. In: International Supercomputing (June 2008)
12. 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., Yelick, K.A.: The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley (2006)
13. Colella, P.: Defining software requirements for scientific computing (2004)
14. Boltzmann, L.: Weitere studien über das Wärmegleichgewicht unter gasmolekülen (further studies on the heat equilibrium of gas molecules). *Wiener Berichte* 66, 275–370 (1872)
15. Qian, Y.H., D'Humières, D., Lallemand, P.: Lattice BGK models for Navier-Stokes equation. *EPL (Europhysics Letters)* 17(6), 479 (1992)
16. Succi, S.: The Lattice Boltzmann Equation: For Fluid Dynamics and Beyond. Numerical Mathematics and Scientific Computation. Clarendon Press (2001)
17. Habich, J., Zeiser, T., Hager, G., Wellein, G.: Performance analysis and optimization strategies for a D3Q19 lattice Boltzmann kernel on nVIDIA GPUs using CUDA. *Advances in Engineering Software* 42(5), 266–272 (2011)

18. Mawson, M., Revell, A.: Memory transfer optimization for a lattice Boltzmann solver on Kepler architecture nVidia GPUs. arXiv preprint arXiv:1309.1983 (2013)
19. Januszewski, M., Kostur, M.: Sailfish: a flexible multi-GPU implementation of the lattice Boltzmann method. ArXiv e-prints (November 2013)
20. Allen, C.B.: An unsteady multiblock multigrid scheme for lifting forward flight rotor simulation. International Journal for Numerical Methods in Fluids 45(9), 973–984 (2004)
21. Allen, C.B.: Parallel universal approach to mesh motion and application to rotors in forward flight. International Journal for Numerical Methods in Engineering 69(10), 2126–2149 (2007)
22. Allen, C.B.: Parallel simulation of unsteady hovering rotor wakes. International Journal for Numerical Methods in Engineering 68(6), 632–649 (2006)
23. Rendall, T.C.S., Allen, C.B.: Unified fluid–structure interpolation and mesh motion using radial basis functions. International Journal for Numerical Methods in Engineering 74(10), 1519–1559 (2008)
24. Allen, C.B., Rendall, T.C.: CFD-based optimization of hovering rotors using radial basis functions for shape parameterization and mesh deformation. Optimization and Engineering 14(1), 97–118 (2013)
25. Herdman, J., Gaudin, W., McIntosh-Smith, S., Boulton, M., Beckingsale, D., Mallinson, A., Jarvis, S.: Accelerating hydrocodes with OpenACC, OpenCL and CUDA. In: High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion, pp. 465–471 (November 2012)
26. Heroux, M.A., Doerfler, D.W., Crozier, P.S., Willenbring, J.M., Edwards, H.C., Williams, A., Rajan, M., Keiter, E.R., Thornquist, H.K., Numrich, R.W.: Improving performance via mini-applications. Sandia National Laboratories. Tech. Rep. (2009)
27. Sandia National Laboratory: The Manteko project home page (February 2014), <http://manteko.org>
28. Mallinson, A.C., Beckingsale, D.A., Gaudin, W.P., Herdman, J.A., Jarvis, S.A.: Towards portable performance for explicit hydrodynamics codes. In: Proceedings of the 1st International Workshop on OpenCL (IWOCL 2013). ACM (May 2013)
29. Saad, Y.: Iterative methods for sparse linear systems. SIAM (2003)
30. Servat, H., Teruel, X., Llort, G., Duran, A., Giménez, J., Martorell, X., Ayguadé, E., Labarta, J.: On the instrumentation of OpenMP and OmpSs tasking constructs. In: Caragiannis, I., et al. (eds.) Euro-Par Workshops 2012. LNCS, vol. 7640, pp. 414–428. Springer, Heidelberg (2013)
31. Komatsu, K., Sato, K., Arai, Y., Koyama, K., Takizawa, H., Kobayashi, H.: Evaluating performance and portability of OpenCL programs. In: The Fifth International Workshop on Automatic Performance Tuning (2010)
32. Rul, S., Vandierendonck, H., D’Haene, J., De Bosschere, K.: An experimental study on performance portability of OpenCL kernels. In: 2010 Symposium on Application Accelerators in High Performance Computing (2010) (papers)
33. Seo, S., Jo, G., Lee, J.: Performance characterization of the NAS parallel benchmarks in OpenCL. In: 2011 IEEE International Symposium on Workload Characterization (IISWC), pp. 137–148. IEEE (2011)
34. Pennycook, S., Hammond, S., Wright, S., Herdman, J., Miller, I., Jarvis, S.: An investigation of the performance portability of OpenCL. Journal of Parallel and Distributed Computing 73(11), 1439–1450 (2013)
35. Du, P., Weber, R., Luszczek, P., Tomov, S., Peterson, G., Dongarra, J.: From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. Parallel Computing 38(8), 391–407 (2012)

36. Cao, C., Dongarra, J., Du, P., Gates, M., Luszczek, P., Tomov, S.: clMAGMA: High performance dense linear algebra with OpenCL. Technical report (lawn 275), ut-cs-13-706, University of Tennessee Computer Science (March 2013)
37. Habich, J., Feichtinger, C., Kostler, H., Hager, G., Wellein, G.: Performance engineering for the lattice Boltzmann method on GPGPUs: Architectural requirements and performance results. ArXiv e-prints (December 2011)
38. Gray, A., Stratford, K.: Ludwig: multiple GPUs for a complex fluid lattice Boltzmann application. In: Couturier, R. (ed.) *Designing Scientific Applications on GPUs*. Chapman & Hall/CRC Numerical Analysis and Scientific Computing Series, Taylor & Francis (2013)
39. Gray, A., Hart, A., Henrich, O., Stratford, K.: Scaling soft matter physics to thousands of GPUs in parallel (2013)
40. Xiong, Q., Li, B., Xu, J., Fang, X., Wang, X., Wang, L., He, X., Ge, W.: Efficient parallel implementation of the lattice Boltzmann method on large clusters of graphic processing units. *Chinese Science Bulletin* 57(7), 707–715 (2012)
41. Geveler, M., Ribbrock, D., Mallach, S., Göddeke, D.: A simulation suite for Lattice-Boltzmann based real-time CFD applications exploiting multi-level parallelism on modern multi- and many-core architectures. *Journal of Computational Science* 2(2), 113–123 (2011)
42. Brandvik, T., Pullan, G.: Acceleration of a 3D Euler solver using commodity graphics hardware. In: 46th AIAA Aerospace Sciences Meeting and Exhibit, January 2008, pp. 607–661 (2008)
43. Elsen, E., LeGresley, P., Darve, E.: Large calculation of the flow over a hypersonic vehicle using a GPU. *Journal of Computational Physics* 227(24), 10148–10161 (2008)
44. Cohen, J., Moalemaker, M.J.: A fast double precision CFD code using CUDA. In: *Parallel Computational Fluid Dynamics: Recent Advances and Future Directions*, pp. 414–429 (2009)
45. Göddeke, D., Buijssen, S., Wobker, H., Turek, S.: GPU acceleration of an unmodified parallel finite element Navier-Stokes solver. In: *International Conference on High Performance Computing Simulation, HPCS 2009*, pp. 12–21 (June 2009)
46. Phillips, E.H., Zhang, Y., Davis, R.L., Owens, J.D.: Rapid aerodynamic performance prediction on a cluster of graphics processing units. In: *Proceedings of the 47th AIAA Aerospace Sciences Meeting*, pp. 1–11 (2009)
47. Barnette, D.W., Barrett, R.F., Hammond, S.D., Jayaraj, J., Laros III, J.H.: Using miniapplications in a Manteko framework for optimizing Sandia’s SPARC CFD code on multi-core, many-core, and GPU-accelerated compute platforms. Technical report, Sandia National Laboratories (2012)
48. Mallinson, A., Beckingsale, D., Gaudin, W., Herdman, J., Levesque, J., Jarvis, S.: CloverLeaf: Preparing hydrodynamics codes for Exascale. Cray User Group (CUG), Napa Valley (2013)

Performance Predictions of Multilevel Communication Optimal LU and QR Factorizations on Hierarchical Platforms

Laura Grigori¹, Mathias Jacquelin², and Amal Khabou³

¹ INRIA Paris - Rocquencourt, Paris, France

laura.grigori@inria.fr

² Lawrence Berkeley National Laboratory, Berkeley, USA

mjacquelain@lbl.gov

³ The University of Manchester, Manchester, UK

amal.khabou@manchester.ac.uk

Abstract. In this paper we study the performance of two classical dense linear algebra algorithms, the LU and the QR factorizations, on multilevel hierarchical platforms. We note that we focus on multilevel QR factorization, and give a brief description of the multilevel LU factorization. We first introduce a performance model called Hierarchical Cluster Platform (HCP), encapsulating the characteristics of such platforms. The focus is set on reducing the communication requirements of studied algorithms at each level of the hierarchy. Lower bounds on communication are therefore extended with respect to the HCP model. We then present a multilevel QR factorization algorithm tailored for those platforms, and provide a detailed performance analysis. We also provide a set of performance predictions showing the need for such hierarchical algorithms on large platforms.

Keywords: QR, LU, exascale, hierarchical platforms.

1 Introduction

Numerical algorithms and solvers play a crucial role in scientific computing. They lie at the heart of many applications and are often key to performance and scalability. Due to the ubiquity of multicore processors, solvers should be adapted to better exploit the hierarchical structure of modern architectures, where the tendency is towards multiple levels of parallelism. Thus with the increasing complexity of nodes, it is important to exploit these multiple levels of parallelism even within a single compute node. For this reason, classical algorithms need to be revisited so as to fit modern architectures that expose parallelism at different levels in the hierarchy. We believe that such an approach is mandatory in order to exploit upcoming hierarchical exascale computers at their full potential.

Studying the communication complexity of linear algebra operations and designing algorithms that are able to minimize communication is a topic that has received an important attention in the recent years. The most advanced approach

in this context assumes one level of parallelism and takes into account the computation, the volume of communication, and the number of messages exchanged along the critical path of a parallel program. In this framework, the main previous theoretical result on communication complexity is a result derived by Hong and Kung in the 80's providing lower bounds on the volume of communication of dense matrix multiplication for sequential machines [1]. This result has been extended to parallel machines [2], to dense LU and QR factorizations (under certain assumptions) [3], and then to basically all direct methods in linear algebra [4]. Given an algorithm that performs a certain number of floating point operations, and considering the memory size, the lower bounds on communication are obtained by using the Loomis-Whitney inequality, as for example in [2, 4]. While theoretically important, these lower bounds are derived with respect to a performance model that supposes a memory hierarchy in the sequential case, and P processors without memory hierarchy in the parallel case. Such a model is not sufficient to encapsulate the features of modern hierarchical architectures.

On the practical side, several algorithms have been introduced recently [5, 6, 7, 8]. Most of them propose to use different reduction trees depending on the hierarchy. However, the focus is set on reducing the running time without explicitly taking communication into consideration. In [8], Dongarra et al. propose a generic algorithm implementing several optimizations regarding pipelining of computation, and allowing to select different elimination trees on platforms with two levels of parallelism. They provide insights on choosing the appropriate tree, a binary tree being for instance more suitable for a cluster with many cores, while a flat tree allows more locality and CPU efficiency. However, neither theoretical bounds nor cost analysis are provided in these studies. Moreover, even if cache-oblivious algorithms are natural good candidates for reducing communication requirements at every level, they are not good candidates for large parallel implementations. We thus focus on cache- and parallelism-aware algorithms.

In the first part of this paper we introduce a performance model that we refer to as the Hierarchical Cluster Platform (HCP) model. Provided that two supercomputers might have different communication topologies and different compute nodes with different memory hierarchies, a detailed performance model tailored for one particular supercomputer is likely to not reflect the architecture of another supercomputer. Hence the goal of our performance model is to capture the main characteristics that influence the communication cost of peta- and exa-scale supercomputers which are based on multiple levels of parallelism and memory hierarchy. We use the proposed HCP model to extend the existing lower bounds on communication for direct linear algebra, to account for the hierarchical nature of present-day computers. We determine the minimum amount of communication that is necessary at every level in the hierarchy, in terms of both number of messages and volume of communication. Moreover, to the best of our knowledge, there is currently no algorithm targeting hierarchical platforms with more than two levels, nor any lower bound on communication for such platforms.

In the second part of the paper we introduce a multilevel algorithm for computing the QR factorization (*ML-CAQR*) that is able to minimize the

communication at each level of the hierarchy, while performing a reasonable amount of extra computations. We note that we have also developed two multilevel algorithms for the LU factorization (1D-*ML-CALU* and 2D-*ML-CALU*). However we restrict our study to the QR factorization here. We refer interested readers to the technical report [9] for more details about the multilevel LU algorithms. These recursive algorithms rely on their corresponding 1-level algorithms (resp. *CAQR* and *CALU*) as their base case. Indeed, *CAQR* and *CALU* are known to attain the communication lower bounds in terms of both bandwidth and latency with respect to the simpler one level performance model.

2 Background: The QR Factorization

The QR factorization of an m -by- n matrix is a widely used algorithm, be it for orthogonalizing a set of vectors or for solving least squares problems with m equations and n unknowns, where $m \geq n$. It is known to be an expensive $mn^2 + 1/3n^3 + O(n^2)$, but very stable factorization. It is thus crucial to optimize its performance. The algorithm decomposes an m -by- n matrix A into two matrices Q and R such that $A = QR$, where Q is an m -by- m orthogonal matrix, while the m -by- n matrix R is upper triangular.

The QR factorization is obtained by applying a sequence of m -by- m unitary orthogonal transformations on the input matrix A . An unitary transformation U_i introduces some zeros below the diagonal in the current updated matrix. The two basic transformations are Givens rotations and Householder reflections. A Givens rotation introduces a single zero while a Householder reflection zeroes out every element below the diagonal. Using Givens rotations, disjoint pairs of rows can be processed concurrently. Householder reflections, though not displaying the same parallelism, are less computationally expensive.

Tree-based algorithms intent to benefit from both methods. Householder transformations are applied on local domains, or tiles, before getting eliminated two-by-two in a Givens-like approach. Communication Avoiding QR (*CAQR*) [3] belongs to this category, and organizes the computations so as to match the lower bounds on communication introduced in [4]. After $\min(m, n)$ transformations, the resulting R factor is stored in place in the upper triangular part of matrix A while the matrix Q is assumed to be implicitly stored in the lower triangular part using the compact *WY* representation for Householder reflections [10]. If needed, Q can be retrieved at the cost of extra computations by computing $Q = I - YTY^T$.

3 Toward a Realistic Hierarchical Cluster Platform Model (Hcp)

The focus of this study is set on hierarchical platforms running HPC applications and displaying increasingly deeper hierarchies. Such platforms are composed of two kinds of hierarchies: (1) a network hierarchy composed of interconnected network nodes, stacked on top of a (2) compute nodes hierarchy [11]. This compute

hierarchy can be composed for instance of shared memory NUMA multicore nodes. Moreover, on most modern supercomputers, compute nodes are often grouped into *drawers* displaying higher local communication speeds. Such drawers typically belong to the network hierarchy, which is clearly not only a router hierarchy.

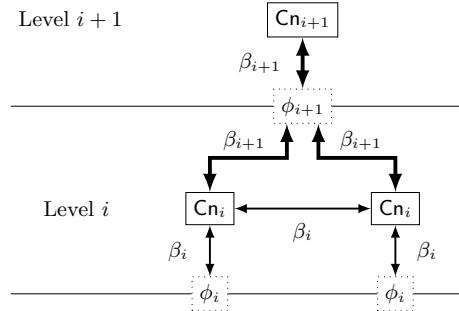


Fig. 1. Components of a level i in the HCP model

The HCP model considers such platforms with l levels of parallelism, and uses the following assumptions. Level 1 is the deepest level in the hierarchy, where actual processing elements are located (for example cores). Each of these processing elements has its own local memory of size M_1 and a computing speed γ . A compute node of level $i + 1$, denoted as $C_{N_{i+1}}$ on Figure 1, is formed by P_i compute nodes of level i (two nodes in our example). The total number of processing elements of the entire platform is $P = \prod_{i=1}^l P_i$, while the total number of compute nodes of level i is $P_i^* = \prod_{j=i}^l P_j$. We let $M_i = M_1 \cdot \prod_{j=1}^{i-1} P_j$ be the aggregated memory size of a node of level $i > 1$.

The network latency α_i and the inverse bandwidth β_i apply throughout an entire level i . Moreover, we assume that generally, the higher in the hierarchy, the more expensive communication costs.

We also consider a message aggregation capacity ϕ_i at each level of the hierarchy, which determines the actual number of messages required to send a given amount of data. We refer to the number of messages sent at level i as S_i , and to the exchanged volume of data as W_i . $\bar{S}_i = S_i \cdot \alpha_i$ is the associated latency cost, while $\bar{W}_i = W_i \cdot \beta_i$ is the bandwidth cost. These notations will be used throughout the rest of the paper.

For the sake of simplicity in both algorithm description and cost analysis, we assume the P_i compute nodes of level i to be virtually organized along a 2D grid topology, that is $P_i = P_{r_i} \times P_{c_i}$ (note that any topology could be mapped onto a 2D grid).

We note that the model makes abstraction of the detailed architecture of a compute node or the interconnection topology at a given level of the hierarchy. Hence such an approach has its own limitations, since the predicted performance might not be accurate. However, while keeping the model tractable, this model better reflects the actual nature of supercomputers than the one level model

assumed so far, and helps to understand the communication bottlenecks of common linear algebra operations. We also note that our model does not apply to platforms with heterogeneity in the processing elements such as GPU and multi-GPU clusters.

Communicating under the Hcp Model. We now describe how communication happens in the HCP model, and how messages flow through the hierarchy. We assume that if a compute node of level i communicates, all of its lower level nodes participate. Hence if some data has to be sent over the network, it first has to be collected from all the cores available on one node. We denote as *counterparts* of a compute node of level i all the nodes of level i lying in remote compute nodes of level $i+1$ and having the same local coordinates. We therefore have the relation $W_i = W_{i+1}/P_i$.

As an example, let us detail a communication of a volume of data W_i taking place between two nodes of level i . A total of P/P_i^* processing elements of level 1 are involved. Each has to send a chunk of data $W_1 = W_i P_i^*/P$. Since this amount of data has to fit in memory, we obviously have $\forall i, M_1 \geq W_1 = W_i P_i^*/P$. These blocks are transmitted to the level above in the hierarchy, i.e. to level 2. A compute node of level 2 has to send a volume of data $W_2 = P_1 W_1$. Since the aggregation capacity at level 2 is ϕ_2 , this requires (W_2/ϕ_2) messages. The same holds for any level k such that $1 < k \leq i$, where data is forwarded by sending (W_k/ϕ_k) messages. We therefore have the following costs:

$$\bar{W}_k = \frac{W_i P_k^*}{P_i^*} \cdot \beta_k, \quad \bar{S}_k = \frac{W_k}{\phi_k} \cdot \alpha_k = \frac{W_i P_k^*}{\phi_k P_i^*} \cdot \alpha_k.$$

This “regular” communication pattern is often encountered in HPC applications, the main target of the HCP model, and is simpler than a purely heterogeneous pattern (which could be encountered in grid environments for instance). Moreover, this organization allows to aggregate data at the algorithm level rather than relying on the actual network topology.

It is interesting to note that the HCP model allows to model several types of networks, depending on their aggregation capacity. We defined the three following network types to demonstrate HCP versatility:

- *Fully-pipelined networks*, aggregating all incoming messages into a single message. This case is ensured whenever $\phi_i \geq P_{i-1} W_{i-1}$. Since M_i is the size of the largest message sent at level i , we assume $\phi_i = M_i$. We also assume that all levels below are themselves fully-pipelined. Therefore, the aggregation capacity becomes $\phi_i = M_i = P_{i-1} \phi_{i-1}$.
- *Aggregating networks*, aggregating data up to volume of $\phi_i < M_i$ before sending a message.
- *Forward networks*, where messages coming from lower levels are simply forwarded. For a given level i , it is required that $\phi_i = \phi_{i-1}$: when each sub-node from level $i-1$ sends S_{i-1} messages, the number of forwarded messages is $S_i = P_{i-1} S_{i-1}$.

Based on the two extreme cases, we assume the aggregation capacity ϕ_i to satisfy $\phi_{i-1} \leq \phi_i \leq P_{i-1} \phi_{i-1}$.

An Example of Hierarchical Platform Modeled by Hcp. Consider a distributed memory platform composed of D drawers having N compute nodes apiece. Let each node be a NUMA shared memory machine, with P processors. Within a node, each socket is connected to a local memory bank of size M , thus leading to a total shared memory of size $M \times P$ per node.

Within a drawer, nodes are interconnected with high speed interconnect such as fiber optics, whereas drawers are connected with more classical copper links. Let inter-drawer communication bandwidth and latency respectively be W_{inter} and S_{inter} . Let intra-drawer communications have a bandwidth W_d and a latency S_d . For intra-node communications, we let W_{mem} (resp. S_{mem}) be the bandwidth (resp. latency) to exchange data with memory.

We model this platform in HCP using three levels, with the following characteristics:

# Comp. nodes	Bandwidth	Latency	Memory	Agg. capacity
$P_1 = P$	$W_1 = W_{\text{mem}}$	$S_1 = S_{\text{mem}}$	$M_1 = M$	$\phi_1 = M$
$P_2 = N$	$W_2 = W_d$	$S_2 = S_d$	$M_2 = P_1 M_1$	$\phi_2 = M \cdot P_1$
$P_3 = D$	$W_3 = W_{\text{inter}}$	$S_3 = S_{\text{inter}}$	$M_3 = P_2 M_2$	$\phi_3 \leq M \cdot P_2$

The aggregation capacities are chosen as follows: (1) On such hierarchical platform, a processor is able to transfer, in one message, its entire local bank of memory to another processor within the same compute node. This is ensured by setting ϕ_1 to M . (2) A compute node can transfer its entire shared memory to a remote node in the same drawer in a single message. The aggregation capacity is therefore chosen as $\phi_2 = MP_1$. (3) Finally, at the topmost level, the interconnect generally does not allow for sending the global volume of data coming from all drawers using a single message. The aggregation capacity is thus chosen as $\phi_3 \leq MP_2$.

HCP allows to model typical HPC platforms, giving communication details at each level of the hierarchy. The switch from a shared memory to a distributed memory environment is handled through the choice of the aggregation capacities.

Lower Bounds on Communication. We now introduce lower bounds on communication at every level of the hierarchy. Lower bounds on communication have been generalized in [4] for direct methods of linear algebra algorithms which can be expressed as three nested loops. We refine these lower bounds under our hierarchical model. For matrix product-like problems, at least one copy of the input matrix has to be stored in memory: a compute node of level i thus needs a memory of $M_i = \Omega(n^2/P_i^*)$. Furthermore, the lower bound on latency depends on the aggregation capacity ϕ_i of the considered level i , where a volume W_i needs to be sent in messages of size ϕ_i . Hence the lower bounds on communication at level i :

$$W_i \geq \Omega\left(\frac{\#flops}{\sqrt{\text{memory}}}\right) = \Omega\left(\frac{n^2}{\sqrt{P_i^*}}\right) \quad (1)$$

$$S_i \geq \Omega\left(\frac{W_i}{\phi_i}\right) = \Omega\left(\frac{n^2}{\phi_i \sqrt{P_i^*}}\right) \quad (2)$$

Note that, for simplicity, we expressed the bound on latency with respect to ϕ_i for each level i . Since we consider $\phi_1 = M_1$, the lower bound on latency for level 1 can also be expressed as $\bar{S}_1 = \Omega(\sqrt{P})$.

4 Multilevel QR Factorization

In this section we introduce *ML-CAQR*, a multilevel algorithm for computing the QR factorization of a dense matrix A . This multilevel algorithm heavily rely on its 1-level communication optimal algorithm *CAQR*, and can be seen as a recursive version of this algorithm. *ML-CAQR* recursive layout naturally allows for local elimination trees adapted to fit hierarchical platforms, thus reducing the communication needs at each level of the hierarchy. *ML-CAQR* is targeting large scale hierarchical platforms. The focus is set on keeping the communication requirements as low as possible at every level of the hierarchy, like *CAQR* on platforms with one level of parallelism.

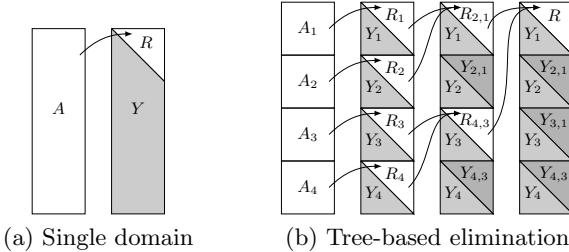


Fig. 2. Structure of the Householder reflectors

ML-CAQR, given in Algorithm 1, uses a recursive tree-based elimination scheme based on Householder reflections. As a tree-based algorithm, *ML-CAQR* stores the Householder reflectors in the lower triangular part of matrix A using a tree structure as in [3]. A small example is depicted on Figure 2, where a panel of matrix A is first split into four domains which are independently factored, then eliminated two by two. The resulting Householder reflectors should be applied following the same order to reflect the update of this panel.

At the topmost level of the hierarchy, *ML-CAQR* factors the entire input matrix A panel by panel. A panel is processed in multiple elimination steps following a tree-based approach. At the leaves of the tree, rectangular blocks are factored. The obtained R factors are then grouped two-by-two and eliminated in a sequence of elimination of size $2b_l$ -by- b_l , where b_l is the panel size. Each factorization or elimination corresponds to a recursive call to *ML-CAQR* on the next lower level. After panel factorization, Householder reflectors are sent to remote compute nodes so as to update the trailing matrix using two recursive routines: *ML-Fact* and *ML-Elim*.

When called on two aggregated R factors, *ML-CAQR* and *ML-Elim* take this specific shape into account and do not perform any unnecessary computations.

Algorithm 1. *ML-CAQR(A, m, n, r, P).*

Input: Matrix A , m is the number of rows of A , n is the number of columns, r is the level of recursion, P is the current compute node

Output: Factored matrix with R in the upper triangular part and the Householder reflectors Y in the lower triangular part

```

if  $r = 1$  then
    Call CAQR(A, m, n, b1, P)
else
    for  $kk \leftarrow 1$  to  $n$ , with step of  $b_r$  do
        for Compute nodes  $p \leftarrow 1$  to  $P_{r,r}$  in parallel do
             $h_p \leftarrow \max(b_r, (m - kk + 1)/P_{r,r})$ 
            if  $kk + (p - 1)h_p < n$  then
                panel  $\leftarrow A(kk + (p - 1)h_p : kk + p \cdot h_p - 1, kk : kk + b_r - 1)$ 
                Call ML-CAQR(panel, hp, br, r - 1, p)

        if There are multiple  $R$  factors then
            for  $j \leftarrow 1$  to  $\log P_{r,r}$  do
                Nodes ( $p_{\text{source}}, p_{\text{target}}$ ) used to perform the elimination.
                Send local  $b_r$ -by- $b_r$  to the remote node  $p_{\text{target}}$ 
                Stack two  $b_r$ -by- $b_r$  upper triangular matrices in  $RR$ 
                Call ML-CAQR(RR, 2br, br, r - 1, psource)
                Call ML-CAQR(RR, 2br, br, r - 1, ptarget)

        for Compute nodes  $p \leftarrow 1$  to  $P_{r,r}$  in parallel do
            Broadcast Householder vectors along processor row
            for Compute node  $rp \leftarrow 2$  to  $P_{c,r}$  on same row as  $p$  do
                Call ML-Fact(r - 1, rp)

    if There are multiple  $R$  factors then
        for  $j \leftarrow 1$  to  $\log P_{r,r}$  do
            Nodes ( $p_{\text{source}}, p_{\text{target}}$ ) used to perform the elimination.
            for Nodes  $rp \leftarrow 2$  to  $P_{c,r}$  on same row as  $p_{\text{source}}$  in parallel do
                Remote node  $rp_{\text{target}}$  is on same row as  $p_{\text{target}}$  and same column than  $rp$ 
                 $rp$  sends its local  $A$  to  $rp_{\text{target}}$ 
                Call ML-Elim(r - 1, rp)
                Call ML-Elim(r - 1, rp_{\text{target}})

```

However, for the sake of simplicity, this special case is not taken into account in Algorithm 1.

More formally, for each recursion level r , let b_r be the block size, and s be the internal computation step (it is incremented by b_r).

For each panel of size b_r , *ML-CAQR* proceeds as follows:

1. The panel is factored by using a reduction operation, where *ML-CAQR* is the reduction operator. With a binary tree, it processes as follows:
 - (a) First, the panel is divided into $P_{r,r}$ subdomains of size $(m - s + 1)/P_{r,r}$ -by- b_r , which are recursively factored with *ML-CAQR* at level $r - 1$. At the deepest level, *CAQR* is called.
 - (b) The resulting b_r -by- b_r R factors are eliminated two-by-two by *ML-CAQR* at level $r - 1$, requiring $\log P_{r,r}$ steps along the critical path.

The computation is redundantly performed on each pair of processors as it simplifies the communication pattern.

2. The current trailing matrix is then updated:

(a) Householder reflectors in lower trapezoidal part of the panel have to be broadcasted along processor rows.

(b) Updates corresponding to factorizations at the leaves of the tree are applied using the *ML-Fact* routine.

ML-Fact broadcasts P_{r_r} blocks of Householder reflectors of size $(m - s + 1)/P_{r_r}$ -by- b_r from the column of nodes holding the current panel along rows of compute nodes. At the deepest level, the update corresponding to a leaf is applied as in *CAQR* (see [3]).

(c) The updates due to the eliminations of the intermediate R factors are then applied to the trailing matrix using the *ML-Elim* procedure. Blocks of size b_r -by- $(n - s - b_r + 1)/P_{c_r}$ are exchanged within a pair of compute nodes. At the lowest level, a partial update is locally computed before being independently applied onto each processing elements (similarly to *CAQR*).

5 Multilevel QR Performance Model

In this section, we provide cost analysis of *ML-CAQR* algorithm with respect to the HCP model. Two types of communication primitives are used, namely point-to-point and broadcast operations. To simplify the analysis, we define two recursive costs corresponding to these communication patterns.

In a *point-to-point communication*, a volume D is transferred between two compute nodes of level r . All compute nodes from level 1 to level $r - 1$ below those two nodes of level r are involved, sending their local data to their respective counterparts in the remote node of level r . The associated communication costs are therefore:

$$\bar{W}_{\text{P2P}}(1 \dots r, D) = \sum_{k=1}^r \frac{D \cdot P_k^*}{P_k} \beta_k,$$

$$\bar{S}_{\text{P2P}}(1 \dots r, D) = \alpha_1 + \sum_{k=2}^r \frac{D \cdot P_k^*}{\phi_k P_k^*} \alpha_k.$$

A *broadcast operation* between P_{c_r} compute nodes of level r is very similar to point to point communication. However at every level, a participating node broadcasts its data to P_{c_r} counterparts. A broadcast can thus be seen as $\log P_{c_r}$ point-to-point communications.

We now review the global computation and communication costs of *ML-CAQR*. At each recursion level r , the current panel is factored by doing P_{r_r} parallel calls to *ML-CAQR*. Then, the resulting R factors are eliminated through $\log P_{r_r}$ successive factorizations of $2b_r$ -by- b_r matrices formed by stacking up two upper triangular R factors. Once a panel is factored, the trailing matrix is updated. However, as the Householder reflectors are stored in a tree structure, the updates must be done in the same order as during panel factorizations. These

operations are recursively performed using *ML-Fact* for the leaves and *ML-Elim* for higher levels in the tree. The global recursive cost of *ML-CAQR* is composed of several contributions. We let:

- $T_{CAQR}(m, n, b, P)$ be the cost of factoring a matrix of size m -by- n with *CAQR* using P processors and a block size b .
- $T_{ML-CAQR}(m, n, b, P)$ be the cost of *ML-CAQR* on an m -by- n matrix using P processors and a block size b .
- $T_{P2P}(\text{levels}, \text{volume})$ be the cost of sending an upper triangular R factor within a panel of level r .
- $T_{ML-Fact}(m, n, b, P)$ be the cost of updating the trailing matrix to reflect factorizations at the leaves of the elimination trees.
- Finally, $T_{ML-Elim}(m, n, b, P)$ be the cost of applying updates corresponding to higher levels in the trees.

In terms of communication, *ML-Fact* consists in broadcasting Householder reflectors along process rows, while *ML-Elim* corresponds to $\log P_{r_r}$ point to point communications of trailing matrix blocks between pairs of nodes within a process column. Using these notations, the cost $T_{ML-CAQR}(m, n, b_r, P_r)$ of *ML-CAQR* can be expressed as,

$$\left\{ \begin{array}{ll} \sum_{s=1}^{n/b_r} \left[T_{ML-CAQR}\left(\frac{m-(s-1)b_r}{P_{r_r}}, b_r, b_{r-1}, P_{r-1}\right) \right. \\ \quad + \log P_{r_r} \cdot T_{P2P}(1 \dots r, \frac{b_r^2}{2}) \\ \quad + \log P_{r_r} \cdot T_{ML-CAQR}(2b_r, b_r, b_{r-1}, P_{r-1}) & \text{if } r > 1 \\ \quad + T_{ML-Fact}\left(\frac{m-(s-1)b_r}{P_{r_r}}, \frac{n-sb_r}{P_{c_r}}, b_{r-1}, P_{r-1}\right) \\ \quad \left. + \log P_{r_r} \cdot T_{ML-Elim}\left(2b_r, \frac{n-sb_r}{P_{c_r}}, b_{r-1}, P_{r-1}\right) \right] \\ T_{CAQR}(m, n, b_1, P_1) & \text{if } r = 1 \end{array} \right. \quad (3)$$

ML-CAQR uses successive elimination trees at each recursion level r , each of which are traversed in $\log P_{r_r}$ steps. Moreover, successive trees from level l down to level r come from different recursive calls: they are inherently sequentialized. Thus, the total number of calls at a given recursion level r can be upper-bounded by $N_r = 2^{l-r} \prod_{j=r}^l \log P_{r_j}$. An upper bound on the global cost of *ML-CAQR* can be expressed in terms of number of calls at each level of recursion, broken down between calls performed on leaves or higher levels in the trees.

In the following γ is the flop rate and $\bar{F}_{ML-CAQR}(n, n)$ is the computational cost of *ML-CAQR* applied to a square matrix of size n . We assume that for each level k , we have $P_{r_k} = P_{c_k} = \sqrt{P_k}$, and that block sizes are chosen to make the additional costs lower order terms, that is $b_k = O(n/(\sqrt{P_k^*} \cdot \prod_{j=k}^l \log^2 P_j))$. Then, by expanding all recursive costs from level l down to level 1, the cost of *ML-CAQR* can be expressed as:

$$\bar{F}_{ML-CAQR}(n, n) \leq \frac{4n^3}{P} \gamma + O\left(\frac{l \cdot n^3}{P \prod_{j=1}^l \log P_j}\right) \gamma \quad (4)$$

$$\bar{W}_{ML-CAQR}(n, n) \leq \frac{n^2}{\sqrt{P}} \left(l \cdot \log P_1 + 4l \cdot \prod_{j=1}^l \log P_j + \log P_l \right) \beta_1 \quad (5)$$

$$+ \sum_{k=2}^{l-1} \frac{(l-k) \cdot n^2}{\sqrt{(P_k^*)}} \left(1 + \frac{2 \prod_{j=k}^l \log P_j}{\sqrt{P_l}} \right) \beta_k + \frac{n^2 \cdot \log P_l}{\sqrt{P_l^*}} \beta_l$$

$$+ O \left(\frac{l \cdot n^2}{\sqrt{P} \log P_l} \cdot \beta_1 + \sum_{k=2}^{l-1} \frac{(l-k) \cdot n^2}{\sqrt{P_k^*} \log P_l} \cdot \beta_k + \frac{n^2}{\sqrt{P_l^*} \log P_l} \cdot \beta_l \right)$$

$$\bar{S}_{ML-CAQR}(n, n) \leq l \cdot \sqrt{P} \cdot \prod_{j=1}^l \log^3 P_j \alpha_1 + \sum_{k=2}^{l-1} \frac{n^2 \cdot (l-k) \log P_k}{\phi_k \sqrt{P_k^*}} \alpha_k \quad (6)$$

$$+ \frac{n^2 \cdot \log P_l}{\phi_l \sqrt{P_l}} \left(1 + \frac{1}{\prod_2^{l-1} \sqrt{P_j}} \right) \alpha_l$$

$$+ O \left(\sqrt{P} \cdot \prod_{j=1}^l \log^2 P_j \alpha_1 + \sum_{k=2}^{l-1} \frac{(l-k) \cdot n^2}{\phi_k \sqrt{P_k^*} \log P_l} \alpha_k + \frac{n^2}{\phi_l \sqrt{P_l} \log P_l} \alpha_l \right)$$

Finally, it is important to note that the recursive nature of *ML-CAQR* can lead to three times more computations than the optimal algorithm (we ignore several lower order terms). This is similar to other recursive approaches [12]. Altogether, *ML-CAQR* allows to reach the lower bounds on communications at all levels of the hierarchy up to polylogarithmic factors. Indeed, choosing appropriate block sizes makes most of the extra computational costs lower order terms while maintaining the optimality in terms of communication. We refer the interested reader to the related research report [9] for more details on these costs.

6 Multilevel LU Factorizations

Here we briefly introduce two variants of a multilevel algorithm for computing the LU factorization of a dense matrix, *ML-CALU*. Both algorithms are recursive. The first variant, *1D-ML-CALU*, follows a uni-dimensional approach where the recursion is applied to the entire panel at each recursive call. The second variant, *2D-ML-CALU*, processes a panel by multiple recursive calls on sub-blocks of the panel followed by a “reduction” phase similar to that of *ML-CAQR*. The base case of both recursive variants is *CALU* [13], which uses tournament pivoting to select pivot rows. *1D-ML-CALU* has the same stability as *CALU*. However, while it minimizes bandwidth over multiple levels of parallelism, it allows to minimize latency only over one level of parallelism. *2D-ML-CALU* which uses a two-dimensional recursive approach, is shown to be stable in practice, and reduces both bandwidth and latency over multiple levels of parallelism. A detailed description, a performance analysis, and a stability study of both algorithms can be found in [9].

We note that similar multilevel approaches can be applied in the context of the communication avoiding rank revealing QR factorization [14], as well as the communication avoiding LU factorization with panel rank revealing pivoting [15].

7 Experimental Results: Performance Predictions

Multilevel communication avoiding algorithms are tailored for large scale platforms displaying a significant gap between processing power and communication speed. The upcoming Exascale platforms are a natural target for these algorithms. We present performance predictions on a sample exascale platform. Current petascale platforms already display a hierarchical nature which strongly impacts the performance of parallel applications. Exascale will dramatically amplify this trend. We plan to provide here an insight on what could be observed on such platforms.

Table 1. Characteristics of NERSC Hopper

Level	Type	#	Bandwidth	Latency
1	2x 6-cores Opterons	12	19.8 GB/s	1×10^{-9} s
2	Hopper nodes	2	10.4 GB/s	1×10^{-6} s
3	Gemini ASICS	9350	3.5 GB/s	1.5×10^{-6} s

As exascale platforms are not available yet, we base our sample exascale platform on the characteristics of the NERSC Hopper [16, 17] petascale platform. It is composed of *Compute Nodes*, each with two hexacore AMD Opteron Magny-cours 2.1GHz processors offering a peak performance of 8.4 GFlop/s, with 32 GB of memory. Nodes are connected in pairs to *Gemini ASICs*, which are interconnected through the *Gemini network* [18, 19]. Detailed parameters of the Hopper platform are presented in Table 1.

Table 2. Characteristics of a sample exascale platform

Level	Type	#	Bandwidth	Latency (formula)	Latency (adjusted)
1	Multi-cores	1024	300 GB/s	1×10^{-10} s	1×10^{-9} s
2	Nodes	32	150 GB/s	1×10^{-7} s	1.2×10^{-7} s
3	Interconnects	32768	50 GB/s	1.5×10^{-7} s	1×10^{-6} s

Our target platform is obtained by increasing the number of nodes at all 3 levels, leading to a total of 1 million nodes. The amount of memory per processing element is kept constant at 1.3 GB. Moreover, exascale platforms are likely to be available around year 2018. Therefore, latencies and bandwidths are derived using an average 15% decrease per year for the latency and a 26% increase for the bandwidth [19, 18].

However, doing so might conduct to latencies so low that electrical signals would have to travel faster than the speed of light in vacuum. This is of course impossible. Therefore, to alleviate this problem, we assume that electrical signal travels at 10% of the speed of light in copper, against 90% in fiber optics. We consider the links within a multicore processor to be made out of copper (at level 1) and the die to be at most 3cm-by-3cm. The links between a group of nodes (i.e. at level 2) are assumed to be based on fiber optics while the interconnect at the last level are assumed to be copper links. Finally, we assume the global supercomputer footprint to be 30m-by-30m. These parameters are detailed in Table 2. We model the platform with respect to the HCP model, and use it to estimate the running times of our algorithms.

We note that in order to assess the performance of multilevel algorithms, costs of state-of-the-art 1-level communication avoiding algorithms need to be expressed with respect to the HCP model. To this end, we assume (1) each communication to go through the entire hierarchy: two communicating nodes thus belong to two distant nodes of level l , hence a bandwidth β_l . (2) Bandwidth is shared among parallel communications.

We evaluate the performance of the *ML-CAQR* algorithm as well as *CAQR* on a square matrix of size $n \times n$, distributed over a square 2D grid of P_k processors at each level k of the hierarchy, $P_k = \sqrt{P_k} \times \sqrt{P_k}$. In the following, we assume all levels to be *fully-pipelined*. Similar results are obtained regarding *forward* hierarchies, which is explained by the fact that realistic test cases are not latency bound, but are mostly impacted by their bandwidth cost.

The larger the platform is, the more expensive the communication becomes. This trend can be illustrated by observing the communication to computation ratio, or *CCR* of an algorithm. In Figures 3 and 4, we plot the *CCR* of *CAQR* and *ML-CAQR* on the exascale platform. The shaded areas correspond to

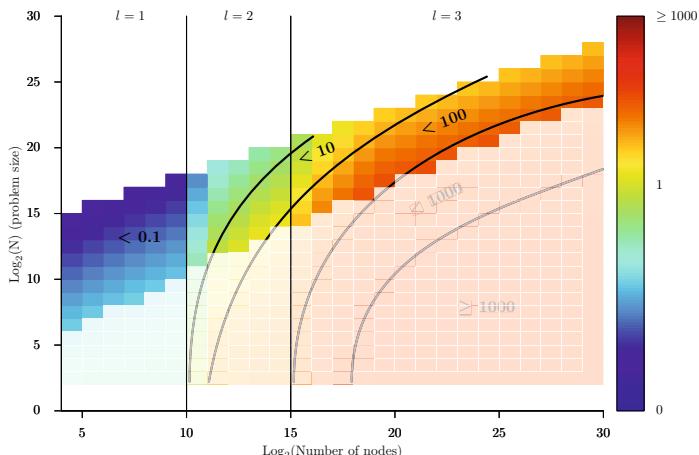


Fig. 3. Prediction of communication to computation ratio on an exascale platform for 1-level *CAQR*

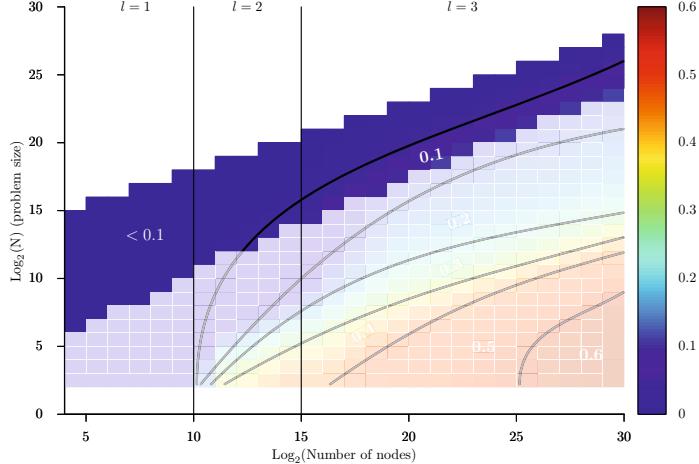


Fig. 4. Prediction of communication to computation ratio on an exascale platform for 1-level *ML-CAQR*

unrealistic cases where there are more processing elements than matrix elements and should not be considered. As the number of processing elements increases, the cost of *CAQR* (in Figure 3) gets dominated by communication. Our multilevel approach alleviates this trend, and *ML-CAQR* (in Figure 4) allows to decrease communication, especially when the number of levels involved is large. Note that for $l = 1$, *ML-CAQR* and *CAQR* are equivalent.

However, as *ML-CAQR* performs more computations than *CAQR*, we compare the expected running times of both algorithms. Here, we denote by running

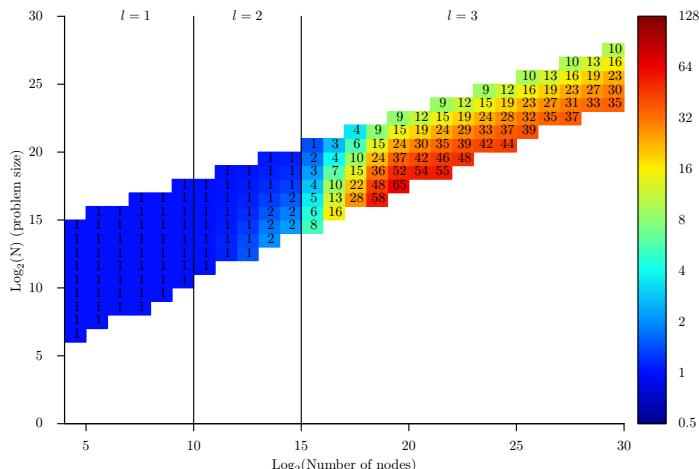


Fig. 5. Speedup of *ML-CAQR* vs. 1-level *CAQR*

time the sum of computational and communication costs. We thus assume no overlap between computation and communication. The ratio of the *ML-CAQR* running time over *CAQR* is depicted in Figure 5. *ML-CAQR* clearly outperforms *CAQR* when using the entire platform, despite its higher computational costs. As a matter of fact in this regime, the running time is dominated by the bandwidth cost, and *ML-CAQR* significantly reduces it at all levels.

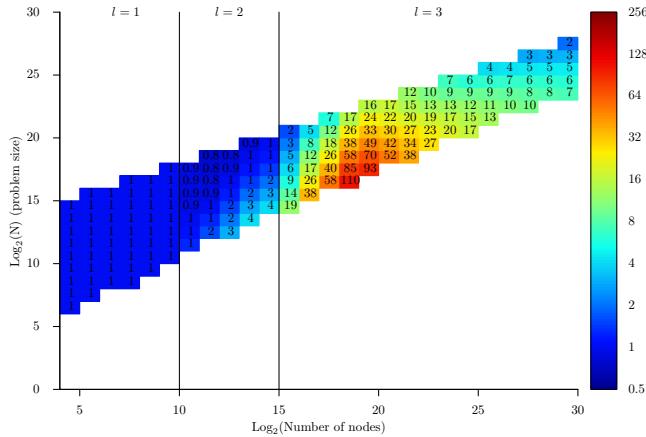


Fig. 6. Speedup of *ML-CALU* vs. 1-level *CALU*

Regarding the running times ratio, depicted in Figure 6, we can also conclude that *ML-CALU* is able to keep communication costs significantly lower than *CALU* when the entire platform is used, leading to significant speedups.

8 Conclusion

In this paper we have studied *ML-CAQR*, an algorithm that minimizes communication over multiple levels of parallelism at the cost of performing redundant computation. The complexity analysis is performed within HCP, a model that takes into account the communication cost at each level of a hierarchical platform. The multilevel QR factorization algorithm has similar stability properties to classic algorithms. Two variants of the multilevel LU factorization have been introduced but not discussed in details. Our performance predictions on a model exascale platform show that for strong scaling, the multilevel algorithms lead to important speedups compared to algorithms minimizing communication over only one level of parallelism.

Acknowledgments. This work was supported in part by the European Research Council Advanced Grant MATFUN (267526) and the Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research and Basic Energy Sciences.

References

1. Hong, J.W., Kung, H.T.: I/O complexity: The Red-Blue Pebble Game. In: STOC 1981: Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing, pp. 326–333. ACM, New York (1981)
2. Irony, D., Toledo, S., Tiskin, A.: Communication lower bounds for distributed-memory matrix multiplication. *J. Parallel Distrib. Comput.* 64(9), 1017–1026 (2004)
3. Demmel, J.W., Grigori, L., Hoemmen, M., Langou, J.: Communication-optimal parallel and sequential QR and LU factorizations. *SIAM Journal on Scientific Computing* (2012), short version of technical report UCB/EECS-2008-89 from 2008
4. Ballard, G., Demmel, J., Holtz, O., Schwartz, O.: Minimizing communication in numerical linear algebra. *SIAM Journal on Matrix Analysis and Applications* 32, 866–901 (2011)
5. Agullo, E., Cotti, C., Dongarra, J., Herault, T., Langou, J.: QR factorization of tall and skinny matrices in a grid computing environment. In: The 24st IEEE Int. Parallel and Distributed Processing Symposium, IPDPS 2010 (2010)
6. Song, F., Ltaief, H., Hadri, B., Dongarra, J.: Scalable tile communication-avoiding QR factorization on multicore cluster systems. In: The 2010 ACM/IEEE Conference on Supercomputing, SC 2010. IEEE Computer Society Press (2010)
7. Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Haidar, A., Herault, T., Kurzak, J., Langou, J., Lemarinier, P., Ltaief, H., Luszczek, P., YarKhan, A., Dongarra, J.: Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA. In: 12th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC 2011) (2011)
8. Dongarra, J., Faverge, M., Hrault, T., Jacquelin, M., Langou, J., Robert, Y.: Hierarchical QR factorization algorithms for multi-core clusters. *Parallel Computing* 39(4-5), 212–232 (2013)
9. Grigori, L., Jacquelin, M., Khabou, A.: Multilevel communication optimal LU and QR factorizations for hierarchical platforms. *CoRR* abs/1303.5837 (2013)
10. Schreiber, R., Van Loan, C.: A storage efficient *WY* representation for products of Householder transformations. *SIAM J. Sci. Stat. Comput.* 10(1), 53–57 (1989)
11. Cappello, F., Fraigniaud, P., Mans, B., Rosenberg, A.: An algorithmic model for heterogeneous hyper-clusters: rationale and experience. *International Journal of Foundations of Computer Science* 16(2), 195–215 (2005)
12. Frens, J., Wise, D.: Qr factorization with morton-ordered quadtree matrices for memory re-use and parallelism. In: ACM SIGPLAN Notices, vol. 38, pp. 144–154. ACM (2003)
13. Grigori, L., Demmel, J., Xiang, H.: CALU: A communication optimal LU factorization algorithm. *SIAM Journal on Matrix Analysis and Applications* 32, 1317–1350 (2011)

14. Demmel, J., Grigori, L., Gu, M., Xiang, H.: Communication avoiding rank revealing QR factorization with column pivoting. Technical Report UCB/EECS-2013-46, EECS Department, University of California, Berkeley (May 2013)
15. Khabou, A., Demmel, J., Grigori, L., Gu, M.: LU factorization with panel rank revealing pivoting and its communication avoiding version. SIAM J. Matrix Analysis Applications 34(3), 1401–1429 (2013)
16. NERSC: Hopper configuration page,
<http://www.nersc.gov/users/computational-systems/hopper/configuration>
17. Shalf, J.: Cray xe6 architecture (2011),
<http://www.nersc.gov/assets/Uploads/ShalfXE6ArchitectureSM.pdf>
18. Editor & lead study, P.K.: Exascale computing study: Technology challenges in achieving exascale systems (2008)
19. Graham, S., Snir, M., Patterson, C.: National Research Council (U.S.). Committee on the Future of Supercomputing: Getting up to speed: the future of supercomputing. National Academies Press (2005)

Hourglass: A Bandwidth-Driven Performance Model for Sorting Algorithms

Doe Hyun Yoon* and Fabrizio Petrini

IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598, USA
doehyun.yoon@gmail.com, fpetrin@us.ibm.com

Abstract. We develop a bandwidth-driven performance model (referred to as *Hourglass*) for sorting algorithms. The model quantifies *dominant data movements* inherent to sorting algorithms (e.g., accesses to/from large buffers and network communication) and estimates a lower-bound execution time. We validate the model with parallel radix sort and merge sort as well as multinode sample sort on leadership high-performance IBM architectures.

The model helps better understand the inherent bottlenecks in a sorting algorithm – the users can leverage this model to optimize software, redesign the algorithm, and/or analyze architectural what-if scenarios to explore innovative designs.

Keywords: sort, performance model, bandwidth.

1 Introduction

The amount of data in our world is exploding, and analyzing and organizing large data sets is becoming a key basis of competition, underpinning new waves of productivity growth, innovation, and scientific opportunities. Sorting algorithms are an essential building block to analyze data and are commonly used by many applications. As a consequence the huge demands on the processing systems translate directly into a renewed need for efficient and scalable sorting algorithms and for innovative architectures that can efficiently support them.

1.1 Sorting in Data-Intensive Computing

Sorting is one of key kernel algorithms in almost all data-processing applications. For the last few decades, researchers have made an extensive effort on analyzing sorting algorithms [1, 2], proposing new ones [3–5], and optimizing performance on various platforms [6–10]. Yet, it is still important, and it is going to be more and more challenging in new data-intensive computing era.

Why Do We Need an Accurate Performance Model? Having a clear performance abstraction of both algorithms and architectures is a key aspect to achieve optimality

* The author is now at Google Inc. New York, NY.

in high-performance systems. Most existing algorithmic design uses simple complexity models, counting operations (e.g., comparisons between elements). The reality of processor architecture forces us to consider the complexity of the memory hierarchy and the communication sub-system as predominant factors in the overall application performance to better understand sorting algorithms, to identify inherent bottlenecks, to guide optimization, and to design better algorithms and architectures.

A Bandwidth-Driven Performance Model. We develop a bandwidth-driven performance model for sorting algorithms, which we refer to as *Hourglass*. We first quantify *dominant data movements* inherent to the algorithms, within and across processing nodes, and identify computing resources that *limit* the data movements, estimating a lower-bound execution time (i.e., an upper-bound performance). With a proper choice of dominant data movements and limiting resources, the estimated execution time can be a tight lower-bound.

We first develop models for single-node parallel sorting algorithms (least significant digit radix sort (LSD) and merge sort) and validate them on two leadership high-performance computing platforms (BlueGene/Q (BG/Q) and Power7 IH (P7IH)) to show the accuracy of the proposed, yet very simple, performance model. We also apply the model to MPI-based multi-node sample sort on a cluster of nodes.

1.2 Contributions

The paper provides several primary contributions.

- A simple and accurate *Hourglass* performance model that captures the essential characteristics of single-node parallel sort.
- The *Hourglass* model is seamlessly extended to multi-node cluster sample sorting.
- An extensive performance evaluation on two leadership high-performance IBM systems, BG/Q and P7IH, showing the robustness of the approach. The Hourglass model is able to capture the relevant aspects of BG/Q and P7IH that have widely different architectural characteristics.
- We provide insight on several aspects of the memory hierarchy, such as fetch-on-write miss policy and huge page support.
- An overall performance analysis that identifies critical trade-offs in the architectural design with respect to the balance of the memory and network bandwidth.

We believe that our analysis provides an important contribution to the design of practical sorting algorithms and is able to identify the architectural balance between memory and network bandwidth to design balanced architectures for high-performance analytics.

2 Related Work

There has been an extensive amount of prior work on performance models of sorting algorithms.

Simplest ones estimate asymptotic computational complexity using the *Big O* notation: comparison-based sort requires $O(N \log N)$ operations, while integer sort is with $O(N)$ operations.

This simple model can be combined with abstract machine models [11, 1] to better identify algorithmic complexity. For example, parallel disk model (PDM) [1] focuses on data movements, instead of computations, estimating the order of I/O operations in a two-level memory hierarchy.

All these models provide only a rough order of complexity (either computation or data movement). They help understand algorithms, but do not estimate real performance (e.g., execution time).

Our model, compared to prior work, estimates a lower-bound execution time. It not only helps understand the inherent bottlenecks in algorithms, but also sets an optimization goal and allows to easily compare different algorithms and architectures. In other words, our model sets a best possible constant in the complexity estimate using the Big O notation.

More detailed performance models can be found in many sorting papers [8, 10]. These models accurately estimate execution time, but their usage is limited to their own software implementations on the target platform. Also, it is unclear whether the achieved performance/efficiency is optimal and whether the derived characteristics are a by-product of a specific software technique, which can be avoided in a better implementation.

Other related work includes networked parallel computation models (bulk synchronous [12], LogP [13, 2], and LogGP [14]) and a roofline performance model [15] that identifies compute and memory-bound characteristics of applications.

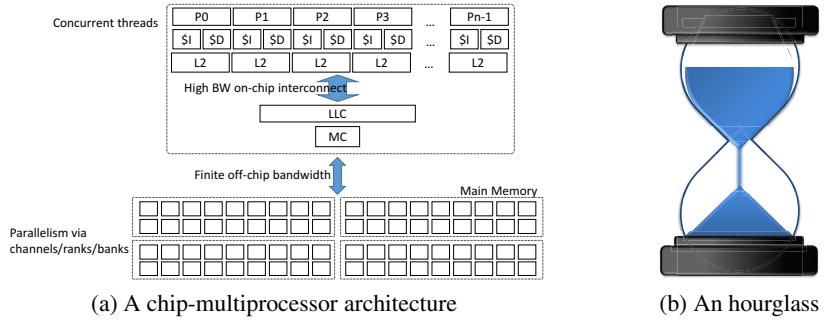
3 A Bandwidth-Driven Performance Model

We develop a bandwidth-driven performance model, based on the data-intensive nature of sorting algorithms and recent trend in computing platforms, described below.

- Sorting (and many other algorithms) is data-intensive. It streams in and out a humongous amount of data and does a relatively small amount of computation; i.e., there is no temporal locality (there are exceptions, though).
- Recent advances in chip multiprocessors (CMP) and parallel processing shifted the performance bottlenecks (of data-intensive applications) from computation and caches to off-chip memory channels, I/O, and network.

Our model first quantifies *dominant data movements* inherent to the algorithm (not something from a specific software implementation) and identifies computing resources that *limit* the data movements, estimating a lower-bound execution time (i.e., an upper-bound performance).

For example, concurrent threads in a typical CMP architecture, shown in Figure 1(a), share a memory channel, and so do memory ranks and banks. No matter how we utilize these parallel resources, applications like sorting *must* transfer a large amount of data through the narrow memory channel.

**Fig. 1.** A chip-multiprocessor example and an analogy to an hourglass

An optimized software will saturate the limited channel bandwidth – the amount of data (from an algorithm) and the bandwidth (for the target platform) determine a lower-bound execution time. We refer to this performance model as *Hourglass*; an hourglass (shown in Figure 1(b)) measures *time*; the amount of sand and the flow rate determine the time, and other details (e.g., the depth in the upper reservoir) are irrelevant.

Note that we use an amount of data movements (instead of operations) and bandwidth (instead of latency). Modern computing systems are highly elaborated to hide latencies (e.g., out-of-order processing), and every computing platform uses different designs. Hence, it is nearly impossible to build a sufficiently accurate and, at the same time, portable model using individual operations and their latencies.

In the remainder of this section, we focus on single-node shared-memory parallel sorting algorithms. The dominant data movements are off-chip memory traffic, which we can easily estimate from an algorithm, and the limiting factor is off-chip memory bandwidth of a target platform, which we can measure using simple microbenchmarks.

3.1 Least Significant Digit Radix Sort

LSD (least significant digit) radix sort (RS) is an integer sorting algorithm. LSD RS is (relatively) simple and easy to parallelize, yet it yields high performance with its $O(N)$ complexity. We use the symbols and definitions described in Table 1 throughout this paper.

Table 1. Nomenclature for sorting algorithms

Symbol	Definition	Default value in this study
K	number of bits per key	64 bits
D	number of bits per radix (or digit)	8 bits
L	sample size in bytes	8 bytes
num_bins	number of bins in histogram (2^D)	256
N	number of keys to be sorted	–

Listing 1.1 is a sequential version of LSD RS. LSD RS has total K/D iterations, processing from the least significant digit (LSD) to the most significant digit (MSD). Each iteration processes a radix (or a digit) and consists of three steps:

- *COUNT* scans input data and generates a histogram w.r.t. the current radix. It sequentially accesses a large array (input data) and randomly updates the histogram (typically fit in L1 cache).
- *PREFIXSUM* scans the histogram array and computes the location of each bin in the output array. *PREFIXSUM* is relatively trivial compared to the other two steps.
- *PERMUTE* reads data from the input array and distributes it to the output array using the *PREFIXSUM* result. It sequentially accesses the input data, but writes are scattered to *num_bins* output streams.

Listing 1.1. Sequential LSD radix sort

```
// sort N 64-bit keys stored in data; buf is an intermediate buffer.
uint64_t *lsd_radix_sort( uint64_t *in, uint64_t N, uint64_t *buf ) {
    uint64_t K = 64, D=8, num_bins = 1UL<<D, msk=num_bins-1;
    uint64_t histo[ num_bins ];

    // total K/D iterations , from LSD to MSD
    for( uint64_t sft=0; sft<K; sft += D ) {
        for( uint64_t bin=0; bin<num_bins; bin++ ) histo[bin] = 0; // init histogram

        // COUNT
        for( uint64_t i=0; i<N; i++ ) histo[ (in[i]>>sft)&msk ]++;

        // PREFIXSUM
        uint64_t sum = 0;
        for( uint64_t bin=0; bin<num_bins; bin++ ) {
            uint64_t prev_sum = sum; sum += histo[bin]; histo[bin] = prev_sum;
        }

        // PERMUTE
        for( uint64_t i=0; i<N; i++ ) buf[ histo[ (in[i]>>sft)&msk ]++ ] = in[i];

        // swap in and out
        uint64_t *temp = buf; buf = in; in = temp;
    }
    return in; // final sorted data
}
```

We parallelize LSD RS using a mechanism similar to Zagha and Blelloch’s work [8]. Parallel LSD RS first partitions the input array to each thread that produces a local histogram of its own data (COUNT). PREFIXSUM then aggregates all local histograms and computes the location of each bin of a thread in the output array so that each thread in parallel PERMUTE has exclusive input and output. In parallel PERMUTE, each thread has *num_bins* sequential output streams, which may appear as random memory accesses at the memory controller (shared among all threads).

Performance Model for LSD Radix Sort. We develop an Hourglass model for LSD RS. First, we quantify the dominant data movements – no matter how we parallelize LSD RS and how we optimize the software, COUNT has to fetch N keys from the input array ($N \times L$ bytes), and PERMUTE has to move N keys from the input to the output ($2 \times N \times L$ bytes).

If we have enough parallel computing resources, COUNT and PERMUTE are bottlenecked by the off-chip channel bandwidth (BW): COUNT takes at least NL/BW , and PERMUTE takes longer than $2NL/BW$.

Bandwidth, however, cannot be modeled using a single number, BW , since modern DDRx memory systems are sensitive to the mix of reads and writes as well as memory access patterns. Hence, we use read-only sequential bandwidth ($BW_{R\text{-seq}}$) for COUNT (since it reads data sequentially) and read-write random-access bandwidth ($BW_{RW\text{-rand}}$) for PERMUTE (since parallel PERMUTE behaves like random memory access as discussed earlier).

Eq. 1 presents the Hourglass performance model for LSD RS. It consists of time to complete COUNT and PERMUTE over K/D iterations: the first term represents COUNT, and the second term for PERMUTE. The estimated execution time is, however, only a lower-bound, since it ignores PREFIXSUM and assumes a maximum off-chip memory channel efficiency.

$$\text{lower_bound} = \frac{K}{D} \times \left(\frac{NL}{BW_{R\text{-seq}}} + \frac{2NL}{BW_{RW\text{-rand}}} \right) = \frac{K}{D} \times NL \times \left(\frac{1}{BW_{R\text{-seq}}} + \frac{2}{BW_{RW\text{-rand}}} \right) \quad (1)$$

With $BW_{R\text{-seq}}$ and $BW_{RW\text{-rand}}$ of a target platform (measured using microbenchmarks), we can estimate the execution time of LSD RS for given K, D, N , and L . Before we validate the Hourglass performance model on BG/Q and P7IH, we first explain the target platforms and their memory systems in the next subsection

Target Platforms and Their Memory Bandwidth. We use a BlueGene/Q (BG/Q) chip and a Power7 IH (P7IH) quad-chip module (QCM) in this study for validating the Hourglass performance model. Table 2 summarizes BG/Q and P7IH.

Table 2. Target architectures

	a BG/Q chip	a P7IH QCM
Core	in-order	out-of-order
Threads/core	4	4
# cores	16 cores per chip	32 cores per QCM
# threads	64 threads per chip	64 threads per QCM ¹
Last-level cache size	32MB	128MB
DRAM capacity	16GB	128GB
Peak memory bandwidth	42.6GB/s	400GB/s

Major differences between P7IH and BG/Q: P7IH and BG/Q provide slightly different computing environments:

¹ P7IH QCM has maximum 128 threads, but, the platform we used activated only 64 threads per QCM.

- P7IH’s caches use fetch-on-write-miss policy [16], while BG/Q does not [17]. The former generates unnecessary reads for sequential write-only patterns.
- While CNK (Compute Node Kernel, the OS for BG/Q) configures BG/Q to avoid address translation penalties using *static memory map* [18], P7IH uses Power ISA’s segmented paging; the default small page configuration yields a poor performance for random accesses.

Microbenchmarks: For the bandwidth-driven performance model, we need memory bandwidth of the target systems. We use memory-intensive microbenchmarks to estimate *attainable* maximum memory bandwidth. To maximize memory channel utilization, we parallelize the microbenchmarks with OpenMP.

- R-Seq: Read a large array to measure read-only bandwidth with sequential access – $BW_{R\text{-seq}}$
- RW-Seq: Copy one large array to another (STREAM copy [19]) to measure read-write bandwidth with sequential access – $BW_{RW\text{-seq}}$
- RW-Rand: Update random memory locations (GUPS [20]) to measure read-write bandwidth with random access – $BW_{RW\text{-rand}}$

On BG/Q, we measured $BW_{R\text{-seq}}$ as 24GB/s, and $BW_{RW\text{-seq}}$ and $BW_{RW\text{-rand}}$ as 28GB/s. Note that BG/Q’s memory system is insensitive to access patterns ($BW_{R\text{-rand}}$ is equal to $BW_{RW\text{-rand}}$) since BG/Q’s static memory map avoids address translation penalties [18], and the memory controller uses closed page policy [21].

On P7IH, $BW_{R\text{-seq}}$, $BW_{RW\text{-seq}}$, and $BW_{RW\text{-rand}}$ are measured as 265GB/s, 270GB/s, and 210GB/s, respectively. P7IH’s memory system, unlike BG/Q, is sensitive to memory access patterns: $BW_{RW\text{-seq}}$ vs. $BW_{RW\text{-rand}}$. To minimize the address translation penalty in random access patterns, we use `hugetlbfs` to enable 16MB page size – we will revisit this issue later.

Evaluation on Blue Gene/Q. We first measure achieved bandwidth w.r.t. the number of threads in COUNT and PERMUTE ($N = 2^{29}$), presented in Figure 2(a). We tried two different compilers (gcc and xlc), simple loop unrolling for COUNT, and -O5 optimization in xlc.

COUNT cannot saturate memory bandwidth with the naïve gcc. With xlc, unrolling, and -O5 option, however, it reaches the goal ($BW_{R\text{-seq}}=24\text{GB/s}$).

PERMUTE, on the other hand, easily achieves the target bandwidth ($BW_{RW\text{-rand}}=28\text{GB/s}$) regardless of build configurations. An optimized version of PERMUTE (xlc.o5.unroll) achieves the goal at only 32 threads. One can expect that target-specific fine-tuning may achieve the same goal using fewer number of threads, but the performance cannot go beyond the bandwidth wall.

We use the best configuration (64 threads, xlc, -O5, and unrolling) to measure the performance of LSD RS. Figure 2(b) compares the measured performance and the Hourglass estimate. In figures, we use sorting rates instead of execution time for a better

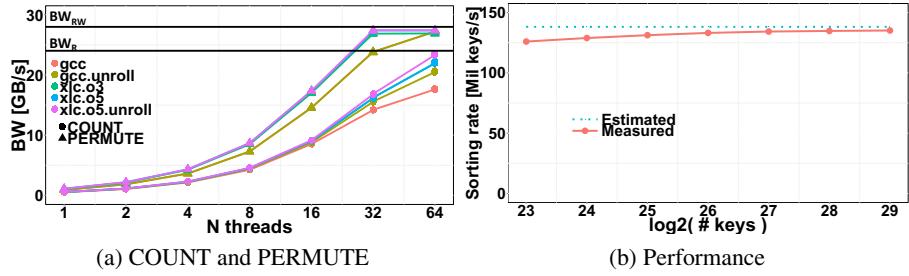


Fig. 2. LSD RS on BG/Q

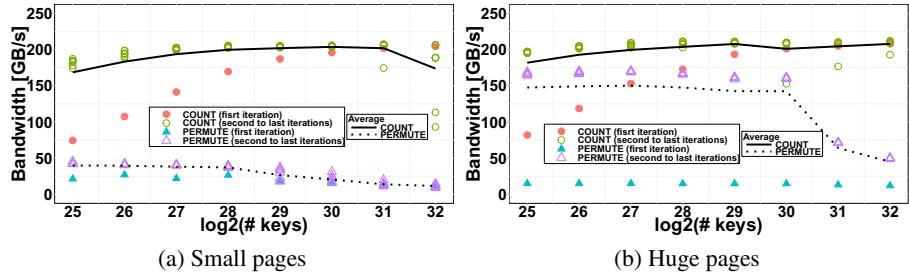


Fig. 3. COUNT and PERMUTE on P7IH

visualization (the higher, the better). The gap between the measured and estimated is below 10%, and decreases as the problem size increases: less than 3% in large problem sizes ($N > 2^{27}$).

Evaluation on P7IH. We measure COUNT and PERMUTE performance using 64 threads with varying problem size (since it affects address translation penalties).² Figure 3(a) shows achieved memory bandwidth in COUNT and PERMUTE when page size is 64kB (default), and Figure 3(b) is with hugetlbf (16MB pages).

COUNT achieves a similar performance with both small and huge pages: around 220GB/s. In general, the larger the problem size, the higher the bandwidth, but only a mild increase. In both page sizes, the first iteration (there are eight iterations) always yields relatively low bandwidth when $N < 2^{31}$. We suspect that this is due to SLB (segment lookaside buffer) warm-up. Note that an SLB miss causes an interrupt to the processor, and the OS fills the SLB entry [16]. In a large problem size (per-thread data $\geq 256\text{MB}$), SLB miss penalty is amortized over accessing 256MB of data.

² We use unrolling in COUNT and gcc, not xlc, since xlc on P7IH did not yield better performance than gcc.

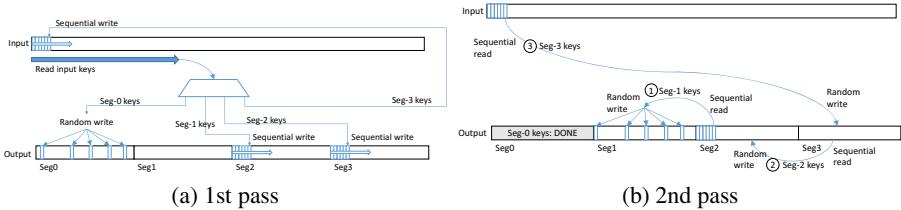


Fig. 4. Segmented PERMUTE with four segments

PERMUTE, on the other hand, is very sensitive to page size. With small pages, it achieves below 50GB/s (much lower than the goal – $BW_{RW\text{-rand}}=210$ GB/s); as discussed earlier, writes in parallel PERMUTE are like random accesses, so most, if not all, write access causes a TLB miss. With huge pages, we can avoid much of TLB misses and achieve relatively high bandwidth (slightly lower than 150GB/s) when $N < 2^{30}$ or 8GB.

Still, there is a nonnegligible gap between the achieved bandwidth and the goal ($BW_{RW\text{-rand}}=210\text{GB/s}$). The reason is P7IH's fetch-on-write-miss policy; every write miss in PERMUTE brings the original (garbage) data from the output array to the cache. This makes PERMUTE to transfer total $3N$ keys (instead of $2N$ keys). Later, we will present a modified model, considering this.

The bandwidth quickly drops to below 70GB/s when $N > 2^{30}$ or 8GB, which is an SLB reach and a TLB reach. PERMUTE randomly accesses the output buffer and thrashes SLB and TLB when $N > 2^{30}$. Later in this section, we will modify LSD RS to gracefully degrade the performance in this case.

Hourglass Model Considering Fetch-On-Write-Miss Policy: To incorporate the effect of fetch-on-write-miss policy in P7IH caches, we modify the Hourglass model (Eq. 1) to Eq. 2; the constant in the second term (time to complete PERMUTE) is changed from 2 to z , where z is 2 with no-fetch-on-write-miss policy (BG/Q) and 3 with fetch-on-write-miss policy (P7IH).

$$\text{lower_bound} = \frac{K}{D} \times NL \times \left(\frac{1}{BW_{\text{R-seq}}} + \frac{z}{BW_{\text{RW-rand}}} \right) \quad (2)$$

Segmented PERMUTE: Poor PERMUTE performance when the problem size is larger than 8GB is a critical issue in P7IH. We propose *segmented PERMUTE* that partitions the output array into a few segments, each of which is 8GB, and writes output to each segment at a time, restricting the range of random access within an SLB/TLB reach. Segmented PERMUTE has more off-chip traffic, lifting the lower-bound execution time, but yields an even tighter lower-bound when SLB/TLB thrashing is common.

Listing 1.2 presents a sequential implementation of segmented PERMUTE (a parallel implementation needs additional operations in PREFIXSUM to enable exclusive outputs in parallel segmented PERMUTE).

Listing 1.2. Sequential segmented PERMUTE

```

// this function can replace the permute in LSD RS
void segmented_permute( uint64_t *in, uint64_t N, uint64_t *out, uint64_t sft,
                        uint64_t msk, uint64_t *histo ) {
    uint64_t seg_size = 8*1024*1024/8; // 8GB segment
    uint64_t seg_sft = log2( seg_size );
    uint64_t num_segs = (N+seg_size-1) / seg_size;

    // set temporary output for second to last segments
    uint64_t *temp[ num_segs ];
    for( uint64_t s=1; s<num_segs-1; s++ ) temp[ s ] = &out[ (s+1)*seg_size ];
    temp[ num_segs-1 ] = &in[ 0 ];

    uint64_t rollback[ num_bins ];
    for( uint64_t bin=0; bin<num_bins; bin++ ) rollback[ bin ] = 0;

    for( uint64_t i=0; i<N; i++ ) {
        uint64_t data = in[ i ];
        uint64_t dest = histo[ (data>>sft)&msk ]++;
        uint64_t seg_id = dest>>seg_sft;
        if( seg_id == 0 ) { // 1st segment: permute to the correct location
            out[ dest ] = data;
        } else { // write out to temporary locations
            rollback[ (data>>sft)&msk ]++;
            *temp[ seg_id ]++ = data;
        }
    }

    // rollback the histo array for those keys sent to temporary locations
    for( uint64_t bin; bin<num_bins; bin++ ) histo[ bin ] -= rollback[ bin ];

    // the second to last segments: permute to the correct locations
    // -- process one segment at a time.
    for( seg=1; seg<num_segs; seg++ ) {
        uint64_t *ptr = &out[ (seg+1)*seg_size ];
        if( seg == num_segs-1 ) ptr = &in[ 0 ];
        for( ; ptr<temp[seg]; ptr++ ) out[ histo[ (*ptr) >>sft ]&msk ]++ = (*ptr);
    }
}

```

We partition the output array into m segments, where each segment is equal to or smaller than the SLB/TLB reach. Segmented PERMUTE first reads N keys from input and permute only those keys belong to the first segment to the output, while the keys belong to the second to last segments are sequentially written out to temporary arrays (Figure 4(a)). It moves $z \times N$ keys in total. Then, it processes each segment, one by one: read the keys from the temporary array and permute them to the correct output locations (Figure 4(b)). There are total $m - 1$ steps, each of which moves N/m keys. Note that segmented PERMUTE does not require additional buffer space for temporary arrays, as shown in Figure 4.

Eq. 3 models the performance of segmented PERMUTE, which adds a third term to Eq. 2, where $m = \lceil \frac{N \times L}{\text{segment_size}} \rceil$. In P7IH, $\text{segment_size} = 8\text{GB}$.

$$\text{lower_bound} = \frac{K}{D} \times NL \times \left(\frac{1}{BW_{\text{R-seq}}} + \frac{z}{BW_{\text{RW-rand}}} + \frac{m-1}{m} \times \frac{z}{BW_{\text{RW-rand}}} \right) \quad (3)$$

LSD Radix Sort Performance: We ran the parallel (64-thread) LSD RS with different configurations and compare them with the Hourglass models in Figure 5.

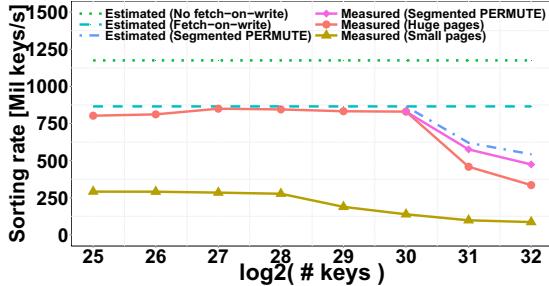


Fig. 5. LSD RS performance on P7IH

LSD RS with small pages, as expected, performs poorly. One with huge pages, with much reduced TLB misses in PERMUTE, performs reasonably close to the model with fetch-on-write-miss, when $N < 2^{30}$. Segmented PERMUTE alleviates SLB/TLB thrashing and has only a very small gap to the Hourglass model.

Note that LSD RS can perform close to the highest upper-bound (the model without fetch-on-write) if we use 1TB segments and 16GB pages³ and avoid fetch-on-write-miss.

3.2 Merge Sort

Merge sort is a comparison-based sort. Its complexity is $O(N \times \log_2 N)$. We estimate a lower-bound execution time of merge sort as in Eq. 4. Note that we use $BW_{\text{RW-seq}}$ instead of $BW_{\text{RW-rand}}$ since merge sort, even a parallel implementation, has good spatial locality.

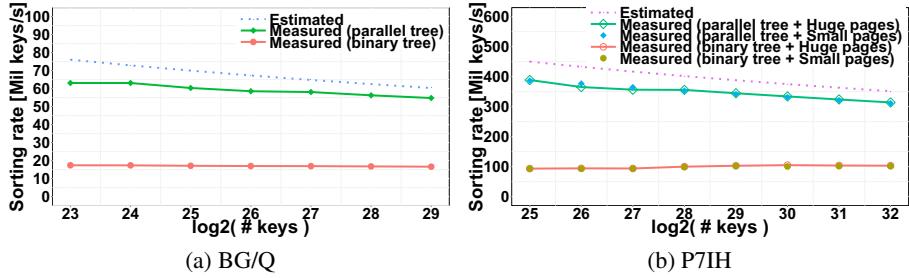
$$\text{lower_bound} = NL \times \log_2 N \times \frac{z}{BW_{\text{RW-seq}}} \quad (4)$$

A naïve merge sort implementation constructs a binary tree; there is a plenty of parallelism initially, but some later iterations cannot be easily parallelized. To overcome this inefficiency, we implement a parallel tree in merge sort as follows: First, the input data is equally divided, and each thread performs sequential merge sort for its own data. Assuming uniform random data, the first thread picks splitters from its sorted data and broadcast to the other threads, followed by parallel merge sort. The second step is alike splitter-based multinode sorting, except data is not explicitly moved across computing nodes.

Figure 6(a) and Figure 6(b) show the estimated and measured performance of merge sort on BG/Q and P7IH, respectively. In both platforms, merge sort with binary tree has poor performance, but the parallel tree implementation yields a performance close to the upper-bound.

An interesting point is that, unlike radix sort, merge sort (in P7IH) is insensitive to page size since both reads and writes are sequential. One may compare merge sort and

³ But, it is available only on AIX [22], while we use Linux on P7IH.

**Fig. 6.** Merge sort performance on BG/Q and P7IH

radix sort using small pages (default in most platforms) and conclude that merge sort is good enough or better than radix sort. With the Hourglass model, we can easily identify the upper-bound in radix sort and avoid this pitfall.

3.3 Caveats

Our model is simple, yet it is accurate and captures the critical traits inherent to the algorithms. As shown with the P7IH case (segmented PERMUTE), it helps elaborate the algorithm to improve the performance. The presented performance model, however, has some caveats:

- This model works only if we have enough parallelism to saturate off-chip memory bandwidth. Some algorithms are inherently difficult to parallelize (e.g., quicksort) and will perform much worse than the Hourglass model estimate.
- If problem size is small enough to fit in the cache, our model is a complete nonsense – but, this tiny, toy problem is none of anyone’s interest.
- In sorting algorithms that reduce working sets as they process data, e.g., quicksort and MSD (Most Significant Digit) radix sort, the Hourglass model works only for the initial steps. Modeling cache-oblivious sorting algorithms (e.g., funnel sort [4]) is challenging, in that respect.

4 Application to Multinode Sorting

We develop the Hourglass performance model to multinode sorting in this section. The major difference to single-node sorting is that keys distributed among multiple nodes need to be shuffled over network, at least once – typically through *all-to-all* communication. Again, an optimized implementation will saturate network bandwidth; hence, we can extend the bandwidth-driven performance model to multinode sorting.

As an example of multinode sorting, we implement MPI-based sample sort as follows (N is the number of keys to be sorted, P is the number of nodes, R is the number of MPI ranks – hence, each node has R/P ranks):

- Step 1: Each MPI rank sorts its own data chunk (use LSD RS); k keys are taken (as samples) at every N/k -th key in the sorted data, and then the sampled keys are exchanged among nodes using MPI_Allgather(). Each node then sorts gathered samples (total $k \times R$ samples) and identify splitters from the sorted samples (every k -th sample).
- Step 2: Exchange keys using MPI_Alltoallv().
- Step 3: Local sorting using LSD RS.

Performance Model. Ignoring operations on small number of samples, we develop an Hourglass performance model for the sample sort in Eq. 5: the major operations in Steps 1 and 3 are LSD RS, which we can estimate using the developed single-node model; and for estimating Step 2, we assume that key values are uniform random (hence, Step 2 does balanced all-to-all communication) and use all-to-all bandwidth, $BW_{\text{all_to_all}}$ (per-node all-to-all bandwidth).

$$\text{lower_bound} = 2 \times \frac{NL}{P} \times \frac{K}{D} \times \left(\frac{1}{BW_{\text{R-seq}}} + \frac{z}{BW_{\text{RW-rand}}} \right) + \frac{NL}{P} \times \frac{1}{BW_{\text{all_to_all}}} \quad (5)$$

All-to-All Bandwidth. We use a microbenchmark to measure $BW_{\text{all_to_all}}$ – each MPI rank allocates 1GB of data, and we measure time to finish MPI_Alltoall function that exchange 1GB data among the other ranks.

We use a P7IH supernode (SN) that has 31 P7IH nodes (QCMs). Each SN consists of four drawers, which has eight nodes (total 32 nodes per SN, but one is reserved for administrative tasks).

Figure 7(a) shows the measured all-to-all bandwidths with the number of nodes varying from 1 to 31. Since only one thread per MPI rank can call MPI functions, increasing the number of ranks per node helps achieve high $BW_{\text{all_to_all}}$, peak at 32 ranks per node.

$BW_{\text{all_to_all}}$ at single node is not network bandwidth; it is intra-node bandwidth. An SN consists of four drawers, but intra-drawer network has higher bandwidth than that of inter-drawer network, so $BW_{\text{all_to_all}}$ among up to eight nodes is higher than those among 16 and 31 nodes.

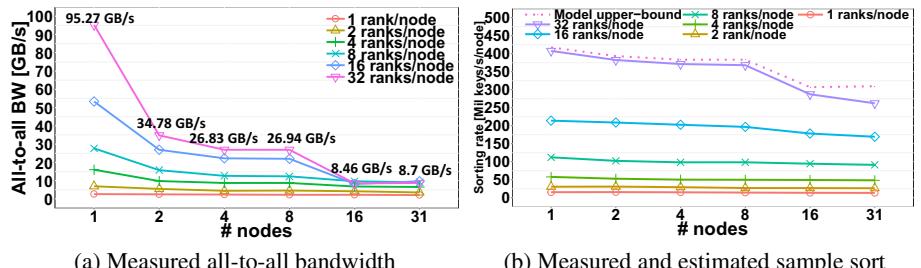


Fig. 7. All-to-all bandwidth and sample sort performance on a P7IH cluster

4.1 MPI-Based Sample Sort

Performance of Sample Sort. Figure 7(b) shows the sample sort performance and the upper-bound estimate (Eq. 5) on a P7IH cluster. As in single-node sorting, the Hourglass model works well in multinode sorting. We did the same experiment on a 32-node BG/Q cluster, where the Hourglass model accurately estimated the performance also. The major sources of errors are as follows:

- With 32 ranks per node, each rank sorts 1GB in Step 1, but our model estimates time to sort 32GB. Sorting 32 independent 1GB data may have better spatial locality in PERMUTE than sorting 32GB.
- Step 3 may have load imbalance since we guess splitters from samples; some nodes may need to sort more than 32GB in total.

We plan to study more sophisticated multinode sorting with possible overlap of all-to-all communication and local sorting.

4.2 Is Multinode Sorting Limited by Memory Bandwidth or Network Bandwidth?

Though network bandwidth is considerably lower than memory bandwidth, in our evaluation, the performance depends more on local sorting – LSD RS needs 8 iterations, while we do all-to-all communication just once. As we try to overlap local sorting and all-to-all, sorting performance will be limited by memory bandwidth.

Multinode sorting, however, can be limited by network bandwidth in a different configuration [23]. So far, we used key-only sorting. Real implementations often sort key-payload tuples. Suppose that a key is 8B, and a payload is 72B; an 80B tuple, 10 times longer than a 64-bit key. In local sort, we can restructure data such that the payload is replaced with a pointer, avoiding expensive payload read/write for intermediate steps. All-to-all communication, however, *must* transfer the large payload anyway – time to complete all-to-all communication is proportional to the tuple size.

Compared to key-only sorting, local sorting time will be doubled with moving both keys and pointers, but all-to-all time will be 10 times longer. Therefore, as the payload size increases, the bottleneck moves from memory bandwidth to network bandwidth, as discussed in CloudRAMSort [9].

Eq. 6 is an Hourglass estimate of sample sort with key-payload tuples (L_k is key size (e.g., 8B), L_x is pointer size (e.g., 8B), and L_p is payload size(e.g., 72B)). Eq. 6 allows us to find a *balanced* design point – a payload size that balances local sort time and all-to-all communication time. Figure 8 shows the balanced design points (payload size) as a function of $BW_{\text{all_to_all}}$: with small payloads (below the plotted line), local sort takes longer than all-to-all; and with large payloads (above the line), all-to-all dominates execution time.

$$\text{lower_bound} = 2 \times \frac{N(L_k + L_x)}{P} \times \frac{K}{D} \times \left(\frac{1}{BW_{\text{R_seq}}} + \frac{z}{BW_{\text{RW_rand}}} \right) + \frac{N(L_k + L_p)}{P} \times \frac{1}{BW_{\text{all_to_all}}} \quad (6)$$

One can use the hourglass model to identify *balanced* design parameters to prevent overprovisioning of resources or to throttle down oversubscribed links for a better energy efficiency.

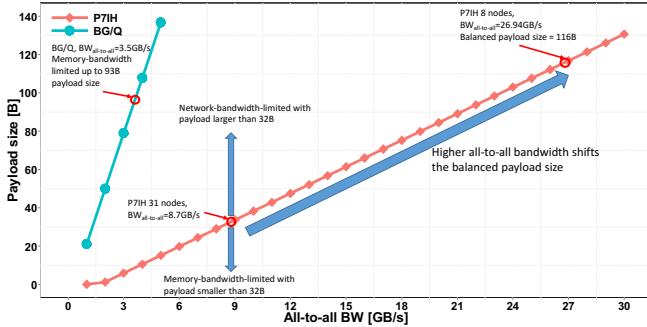


Fig. 8. Balanced payload size as a function of all-to-all bandwidth

5 Discussion

We presented the Hourglass performance model for sorting algorithms and validated the model on leadership IBM computing platforms (BG/Q and P7IH).

The Hourglass model is simple and portable, and the estimated performance is a tight upper-bound. We can use this model to guide software optimization, compare different algorithms and architectures, and find a balanced design point. As future computing platforms are predicted to be more bandwidth limited [24], a bandwidth-driven approach such as Hourglass will be more and more effective in modeling, designing, etc.

We believe that we can extend the Hourglass model to other areas; for instance, external sorting as well as other data-intensive algorithms. In external sorting, the I/O channel is a much more constrained resource than the memory channel, we believe our model will work nicely (perhaps, even better than in-memory sorting) in external sorting. As in Aggarwal and Vitter's disk-based algorithm study [1], the Hourglass model can help algorithms such as FFT, scanning, permute, and search.

References

1. Aggarwal, A., Vitter, J.S.: The input/output complexity of sorting and related problems. *Communications of the ACM* 31, 1116–1127 (1988)
2. Dusseau, A.C., Culler, D.E., Schauer, K.E., Martin, R.P.: Fast parallel sorting under LogP: Experience with the CM-5. *IEEE Transactions on Parallel and Distributed Systems* 7, 791–805 (1996)
3. Arulanandham, J.J., Calude, C.S., Dinneen, M.J.: Bead-Sort: A natural sorting algorithm. *The Bulletin of the European Association for Theoretical Computer Science* (76), 153–162 (2002)
4. Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache-oblivious algorithms. In: Proc. the IEEE Symp. Foundation of Computer Science (October 1999)
5. Bender, M.A., Farach-Colton, M., Mosteo, M.: Insertion sort is $O(n \log n)$. *Theory of Computing Systems* 39(3), 381–397 (2006)
6. Inoue, H., Moriyama, T., Komatsu, H., Nakatani, T.: A high-performance sorting algorithm for multicore single-instruction multiple-data processors. *Software – Practice & Experience* 42(6), 753–777 (2012)

7. Bandyopadhyay, S., Sahni, S.: Sorting on a Cell Broadband Engine SPU. In: Proc. the IEEE Symp. Computers and Communications (ISCC) (July 2009)
8. Zagha, M., Blelloch, G.E.: Radix sort for vector multiprocessors. In: Proc. the ACM/IEEE Conf. on Supercomputing (SC) (November 1991)
9. Kim, C., Park, J., Satish, N., Lee, H., Dubey, P., Chhugani, J.: CloudRAMSort: Fast and efficient large-scale distributed RAM sort on shared-nothing cluster. In: Proc. of the ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD) (May 2012)
10. Satish, N., Kim, C., Chhugani, J., Nguyen, A.D., Lee, V.W., Kim, D., Dubey, P.: Fast sort on CPUs and GPUs: A case for bandwidth oblivious SIMD sort. In: Proc. of the ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD) (June 2010)
11. Alpern, B., Carter, L., Ferrante, J.: Modeling parallel computers as memory hierarchies. In: Proc. Programming Models for Massively Parallel Computers (September 1993)
12. Valiant, L.G.: A bridging model for parallel computation. Communications of the ACM 33, 103–111 (1990)
13. Culler, D., Karp, R., Patterson, D., Sahay, A., Schausler, K.E., Santos, E., Subramonian, R., von Eicken, T.: LogP: Towards a realistic model of parallel computation. In: Proc. the Fourth ACM Symp. Principles and Practice of Parallel Programming (PPoP) (May 1993)
14. Alexandrov, A., Ionescu, M.F., Schausler, K.E., Scheiman, C.: LogGP: incorporating long messages into the logp model — one step closer towards a realistic model for parallel computation. In: Proc. the Seventh Ann. ACM Symp. Parallel Algorithms and Architectures (SPAA) (June)
15. Williams, S., Waterman, A., Patterson, D.: Roofline: An insightful visual performance model for multicore architectures. Communications of ACM 52(4), 65–76 (2009)
16. Sinharoy, B., Kalla, R., Starke, W.J., Le, H.Q., Cargnoni, R., Van Norstrand, J.A., Ronchetti, B.J., Stuecheli, J., Leenstra, J., Guthrie, G.L., Nguyen, D.Q., Blaner, B., Marino, C.F., Retter, E., Williams, P.: IBM POWER7 multicore server processor. IBM Journal of Research and Development 55(3), 1:1–1:29 (2011)
17. Ohmacht, M., Wang, A., Gooding, T., Nathanson, B., Nair, I., Janssen, G., Schaal, M., Steinmacher-Burow, B.: IBM Blue Gene/Q memory subsystem with speculative execution and transactional memory. IBM Journal of Res. and Dev. 57(1/2), 1:1–1:13 (2013)
18. IBM system Blue Gene solution: Blue Gene/Q application development,
<http://www.redbooks.ibm.com/abstracts/sg247948.html>
19. McCalpin, J.: The STREAM benchmark: Computer memory bandwidth,
<http://www.streambench.org/>
20. Earl Joseph II: GUPS (giga-updates per second) benchmark,
<http://www.dgate.org/~brg/files/dis/gups>
21. Team, I.B.G.: Design of the IBM Blue Gene/Q compute chip. IBM Journal of Res. and Dev. 57(1/2), 1:1–1:13 (2013)
22. POWER7 and POWER7+ optimization and tuning guide,
<http://www.redbooks.ibm.com/abstracts/sg248079.html>
23. Sort benchmark home page, <http://sortbenchmark.org/>
24. Rogers, B., Krishna, A., Bell, G., Vu, K., Jiang, X., Solihin, Y.: Scaling the bandwidth wall: Challenges in and avenues for CMP scaling. In: Proc. the 36th Int'l Symp. Computer Architecture (ISCA) (June 2009)

Performance Analysis of Graph Algorithms on P7IH

Xinyu Que, Fabio Checconi, and Fabrizio Petrin

IBM TJ Watson, Yorktown Heights, NY 10598, USA
`{xque, fchecco, fpetrin}@us.ibm.com`

Abstract. IBM Power 775 (P7IH) is the latest supercomputing system that was designed for high-productivity and high-performance. The key innovation on the hub-chip based network makes it perform superior for traditional HPCC benchmarks. In this paper, we detailed characterize the bared network performance with a thin communication stack. Based on that, we present a systematical performance analysis of the data-intensive benchmark, Graph500’s Breadth First Search, on P7IH. We then provide insight into the overall interaction between hardware and software and present the lesson learned on the key bottlenecks of both architecture and data-intensive application.

Keywords: P7IH, Graph Algorithm, Data-intensive Application, Network Performance.

1 Introduction

The Power 775 (P7IH) system [1,2] is IBM’s response to DARPA’s High Productivity Computing Systems (HPCS) Program. The goal is to provide a new generation of economically viable high productivity computing systems for the national security and industrial user community.

The P7IH is specifically designed for high performance, achieving high levels of processing and bandwidth at system scale and has recently attained world record performance numbers for traditional HPCC benchmarks [3]. Unfortunately, there is a lack of analysis of P7IH system on the data intensive applications such as graph algorithms, which are playing an important role in many areas due to the rising Big Data challenges.

In this paper, thus, we focus on understanding the basic network performance on P7IH and overall capability of supporting data intensive graph applications. Specifically, the paper made the following contributions:

1. Detailed analysis on the network characteristics, including latency, bandwidth, routing strategy and flow control on P7IH
2. In-depth examination on the capability of supporting the intrinsic communication need, fine-grained messages with irregular access pattern, for graph applications on P7IH
3. Systematic performance analysis on the Breadth First Search (BFS) kernel in Graph500 on P7IH

The next section walks through the architecture of P7IH system, and graph 500 Breadth First Search kernel. Section 3 examines the low-level network performance

with a rich set of micro-benchmarks, and Section 4 derives several benchmarks to evaluate the unique communication pattern of graph applications. Section 5 provides a vertical analysis of Graph500 BFS kernel on P7IH. Section 6 discusses related work. Finally, we conclude in Section 7.

2 Background

This section first provides an overview of the P7IH system, where we describe the main horse power, POWER 7 chip, interconnection topology and routing mechanism. Then we briefly sketch the Graph 500 Breadth First Search kernel.

2.1 P7IH Architecture

Figure 1 shows the vertical hierarchical structure of the P7IH systems. The P7IH machine is composed of multiple supernodes (SNs). One supernode includes four drawers with each drawer contains eight compute nodes. Each compute node has a Quad-Chip Modules (QCM) and a hubchip. The QCM has four POWER 7 chips and each POWER 7 chip has 8 physical cores with each supporting up to 4-way simultaneous multithread (SMT). Each physical core has independent L1 and L2 caches along with shared L3 cache. Each QCM has eight memory controllers with 128GB memory capacity.

The hubchip is a high radix router which provides network connectivity to the QCMs, which uses $L_{local}(LL)$, $L_{remote}(LR)$, and D links to connect to other compute nodes. To be specific, 7 LL link (24GB/S) is to connect QCMs on the same drawer, and 24 LR link (6GB/S) is to connect the QCMs on the same supernode but not on the same drawer. 16 D link (10GB/S) is to connect the QCMs on other supernodes and there is at least 1 pair of D link between every pair the supernodes, although smaller systems can employ multiple D links between supernode pairs. The LL links are electrical while the LR and D links are optical. The network backbone is a two-level dragonfly, where 32 nodes form into a supernode with all to all connection and every supernode connects to all other supernodes. The hubchip also performs several important I/O functions, including coherence operations, PCI connections to storage adapters, and gateways for global communications.

P7IH supports both direct routing and indirect routing. The direct route employs one of potentially many shortest paths between any two compute nodes in the system. Since there are multiple D links connecting a pair of supernodes, there are multiple shortest paths between two compute nodes. With two-level topology, the longest direct route should have at most three hops with two L hops and one D hop. The indirect routing is supported to guard against potential hot spots in the interconnect. An indirect route is one that involves an intermediate compute node in the route that resides on a different supernode from that of the source and destination compute nodes. An indirect route must contain a shortest path from the source compute node to the intermediate one, and a shortest path from the intermediate compute node to the destination compute node. Thus the longest indirect route should have at most five hops with three L hops and two D hops.

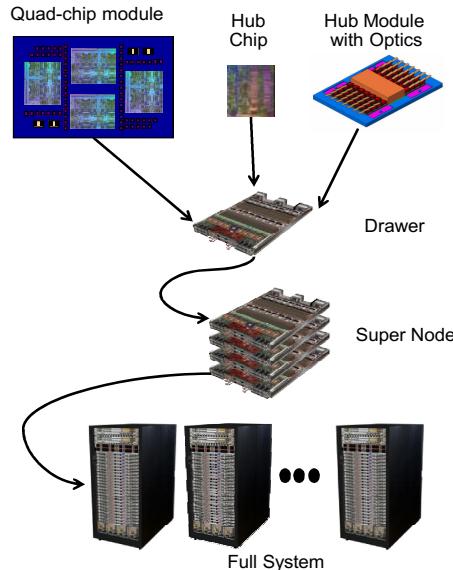


Fig. 1. P7IH Hierarchical Structure

2.2 Graph 500 and Breadth First Search Kernel

Big Data is now everywhere and understanding Big Data can yield advances and intelligence in many areas from oil and gas to healthcare, pharmaceuticals and media. The complexity of the Big Data is normally modeled as graph. Hence, graph algorithms are a core part of such data-intensive analytics workloads in many application areas. Graph 500, introduced in 2010, is a benchmark designed to rank supercomputers to characterize their performance for data-intensive computing workloads, and keep pushing conventional architectures towards their limits. It has initially focused on Breadth First Search (BFS). BFS is an important building block for many sophisticated methods including connected component algorithms [4], and heuristic search algorithms such as A^* [5,6]. The input graph for Graph 500 is generated using R-MAT [7] and the performance is measured by the Traversed Edges Per Second (TEPS).

3 Network Performance Characteristics

Prior to analyzing the performance of graph applications on any complex parallel system, one should understand the factors that may artificially impact performance. The graph applications usually handle large, sparse data sets with relatively simple accompanying computational operations. Thus, the performance is dominated by the parallel system's communication as opposed to the computational capabilities. In this section, we focus on examining the communication capability. To be specific, we use micro-benchmarks to systematically study the low-level network performance.

Experiments Environment. Our micro-benchmarks are based on the HAL/HFI layer communication, which is able to access the low-level Torrents mechanisms with two different flow control mechanisms, namely thread level flow control (Flow control enabled) and no thread level flow control (Flow control disabled). We examined three sets of communication pattern including *point-to-point bandwidth*, *ping-pong latency*, and *all-to-all Bandwidth*. Besides, we also examined the impact of different routing methods to the latency and bandwidth. Considering that the network topology is symmetric and the maximum number of hops between two octants is three, the point-to-point bandwidth and ping-pong latency tests mainly focus on 2 SNs, which cover all the physical links. The all-to-all bandwidth were examined with multiple SNs. Note that on P7IH, there is one node serving as a service node which can not be used by applications. Hence, we can only use 31 compute node on one SN. All the rest tests are executed on a P7IH system running Linux 2.6.32-270.7.7.p7ih.el6.ppc64, which has 8 D links between each supernode pairs.

Point-to-Point Bandwidth: The point-to-point bandwidth is measured between two nodes. All the threads on the sending node send back-to-back messages to the counter part threads on the receiving node and the bandwidth is calculated by dividing the total bytes by the elapsed time.

Ping-pong Latency: The latency tests are carried out in a ping-pong manner with two nodes. In this case, there is only one thread running on each node. The sender thread sends a certain data size message to the receiver thread and waits for an acknowledgement from the receiver. The average one-way latency is calculated.

All-to-all Bandwidth: All-to-all communication is important for graph applications. The intrinsic need of such communication is to fast shuffle the data and serve computation. We mimic this type of communication by having all the threads concurrently sending back-to-back messages to random destinations over multiple iterations. The bandwidth is calculated by dividing the total messages sent per node by the elapsed time.

Point-to-Point Communication. Figure 2 shows the comparison of the point-to-point bandwidth on 2 SNs, where Node 0 is chosen as the sender and we have tuned both number of threads per node and message size. As observed in the figure, the bandwidth increases for large messages. When increasing the threads from 32 to 64, there is marginal improvement on the bandwidth, which may imply that the network may not be able to saturate the higher injection rate. Besides, adding flow control does not incur much penalty for bandwidth with large message. However, there is a noticeable impact for the small message. Last but not least, for different destination, the bandwidth is different which implies the different physical links between the sender and the receiver and we see three different bandwidths.

Figure 3 shows the corresponding latency. Again, we choose node 0 as the sender. As shown in the figure, for large message size (512 bytes and 2032 bytes), the latency looks more stable. While for small messages (8 bytes and 128 bytes), it shows a bit variability. Besides, the difference of the latency implies the distance between the receiver and the sender and large message size emphasizes such difference. For example, in Figure 3 (a) and (b), it shows four different distance including Node 1-6, Node 7-30, Node 31-38,

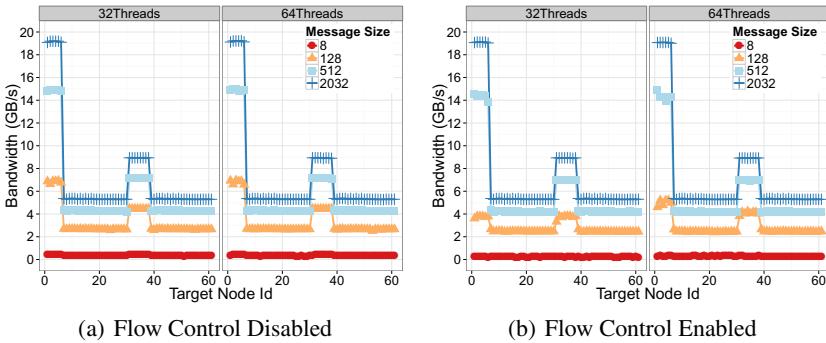


Fig. 2. Point to Point Bandwidth on 2 SN

and Node 39-61. Node 1-30 are the nodes within the same SN as the sender and Node 31-61 are the nodes in the other SN. Considering that there are $8 D$ links between every pair of SNs, Node 31-38 should reside in a drawer which has D links connected to the drawer that Node 0 belongs to.

Figure 4 shows the bandwidth across different physical links, where we had 32 threads running on each node with the message size from 8 bytes to 2032 bytes. The bandwidth increases with the increasing of the message size. Compared to flow control enable case, higher bandwidth is observed for small message without flow control (8 bytes to 256 bytes). When it comes to large message, the bandwidth becomes comparable for two different flow control mechanisms. The measured maximum bandwidths for LL link, LR link, and D link are 19.02 GB/s, 5.2 GB/s, and 8.9 GB/s respectively. Figure 5 shows the latency cross different physical links with different flow control mechanisms. The message size is 8 bytes. For both cases, as shown in the figure, the LL links give the lowest latency and following that is the LR links. Messages crossing D links require higher latency. The results indicate that the communication has lower latency within one SN than between different SNs.

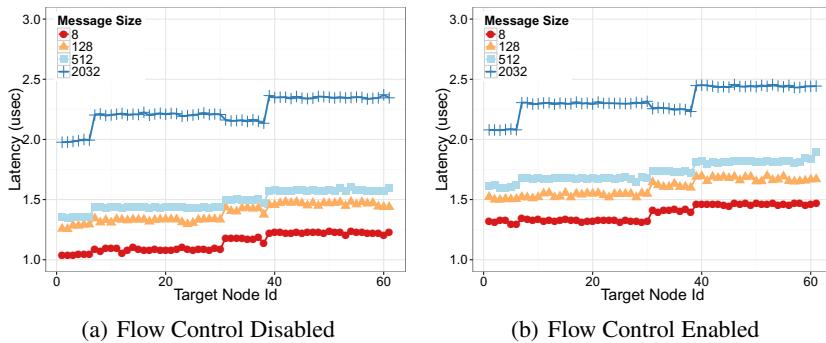
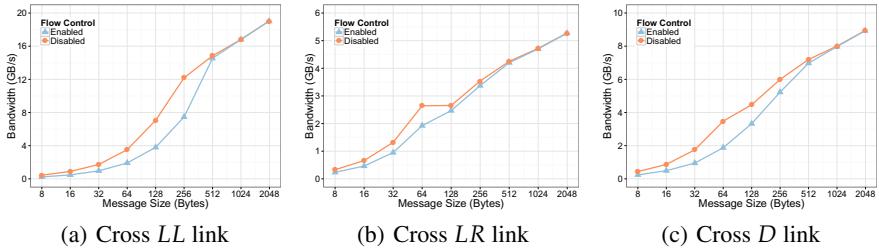
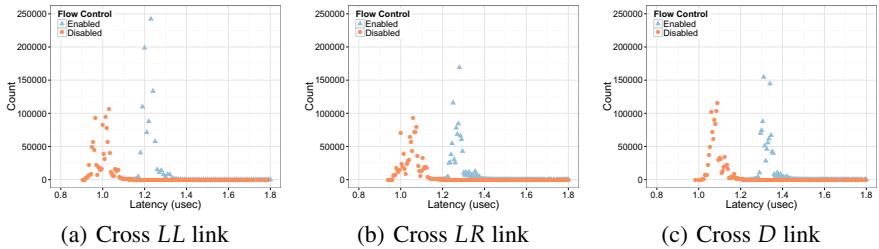
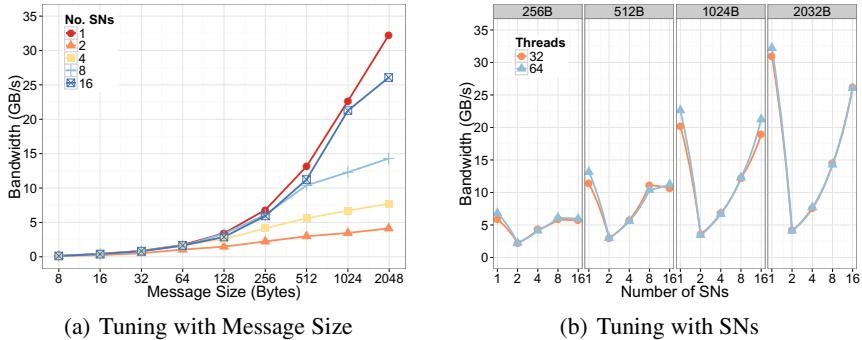


Fig. 3. Ping-pong Latency on 2 SN

**Fig. 4.** Bandwidth Cross Different Links**Fig. 5.** Latency Cross Different Links**Fig. 6.** All to All Bandwidth on Multiple SNs

All-to-all Bandwidth on Multiple SNs. Figure 6 shows the all-to-all bandwidth on multiple SNs. Figure 6 (a) shows the bandwidth trend with different message size, where the bandwidth increases for large messages. Figure 6 (b) shows all-to-all bandwidth over multiple SNs for a fixed message size. The bandwidth has the worst performance on 2 SNs, considering that there are only 8 D links saturating half of the network traffic. When the number of SNs increases, more D links help to saturate the traffic, thereby increasing the all-to-all bandwidth. We observed the bandwidth on 16 SNs is 26.2 GB/s. We expect that the bandwidth can still benefit from more SNs.

Direct Routing vs. Indirect Routing. The P7IH system supports both direct routing and indirect routing. The longest direct route has at most 3 hops($L-D-L$) and the longest

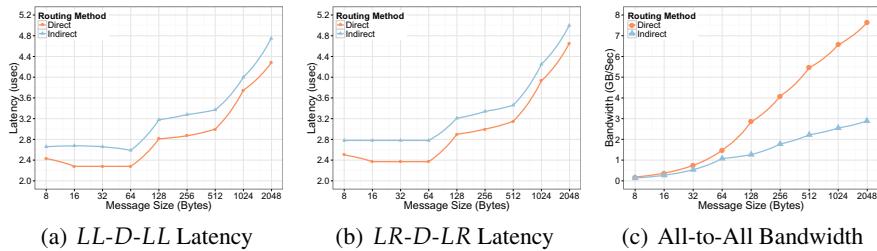


Fig. 7. Performance with Different Routing Method

indirect route has at most 5 hops(*L-D-L-D-L*). To understand the latency and all-to-all bandwidth difference with these two routing methods, we have conducted experiments on 4 SNs. Figure 7 (a) and (b) show the latency comparison , where in (a), **Direct** refers to the direct route crossing *LL-D-LL* links and in (b), **Direct** refers to the direct route crossing *LR-D-LR* links. Hence, the **Indirect** refer to the corresponding indirect routes through an intermediate supernode. As shown in the figure, the latency difference for the two routing methods are consistent (around 400 nanoseconds), which may imply that the time to pass an extra super node is more or less similar. Figure 7 (c) shows the all-to-all bandwidth comparison for different routing methods on 4 SNs. It is not surprising that the bandwidth for indirect routing only reaches half of the bandwidth for direct routing. Because the indirect routing enables a logical all-to-all to reach the intermediate supernode and then performs a second deterministic all-to-all communication.

To summarize, with the thin software stack, the network provides only 80%-87% of the peak point-to-point bandwidth and the corresponding latency is around 1 microseconds. The maximum all-to-all bandwidth obtained on 16 SNs is around 26 GB/s and we have observed that the indirect routing has a noticeable impact on the latency and also decreases the all-to-all bandwidth dramatically.

4 Fine-Grained Communication Capability

As mentioned earlier, graphs offer a natural representation for unstructured data from a variety of applications, which helps to analyze the relationships(edges) between the individual data points(vertices). The analytic in a distributed memory system usually uses Bulk Synchronous Parallel(BSP) model. The performance is dominated by the capability of fine-grained communication and irregular access pattern, which sustain a large number of small (usually 8 or 16 bytes), random remote data access to the memory and interconnect.

In this section, we derived a communication library on top of HAL/HFI layer communication to serve the purpose of supporting the fine-grained communication and irregular access pattern for the BFS kernel. We want to examine the performance can be achieved with this type of communication on P7IH machine and understand how does it different from the network performance we observed in the previous section.

The communication library aggregates small message on a per-destination basis, buffering them until enough data is ready to be sent to each destination. In order to

avoid contention, we used dedicated coalescing queues for each destination on each thread. The amount of data buffered for each destination is one maximum-size packet (2032 bytes). The copies made at the lower levels of the communication stack ensure that, as soon as a `send()` call returns the buffers can be reused (the copy made at low level will be used in case a retransmission is needed).

The optimization of the communication library went through several steps, illustrated in the following paragraphs. We started trying to achieve the best possible rate of injection into the coalescing buffers, defining three incremental micro-benchmarks to evaluate the performance of various stages of the injection process. The benchmarks are defined as follows:

write: for each destination, keep an index of the current position in the buffer; on each injection write eight bytes at the position specified by the index and increment (modulo the size of the buffer) the index position.

inject: same as `write`, but as soon as a packet is ready try to send it over the network, with flow control disabled.

injrec: same as `inject`, but pair each call to `send()` with a call to the polling routine for the communication FIFO. Don't make any attempt to count the received packets or to ensure they have all been received.

The code for the three options is shown in Listing 1.1. The pattern of the destinations for the intended messages was pre-generated randomly and stored in memory.

```

1  /* Write 8 byte entries to a per-destination ring structure; the
2   * destination node is read from a pre-generated pattern.
3   */
4  static void op_write(uint64_t *pattern, uint64_t *index, uint64_t *data)
5  {
6      for (uint64_t i = 0; i < LOOPS; i++) {
7          uint64_t dst = pattern[i % SAMPLES];
8          uint64_t idx = (index[dst]++) % PKT_SIZE;
9          data[dst * PKT_SIZE + idx] = ~0UL;
10     }
11 }
12
13 /* Same as above, plus inject packets into the network as soon as
14 * they are completed.
15 */
16 static void op_inject(int tid, uint64_t *pattern, uint64_t *index, uint64_t *data)
17 {
18     for (uint64_t i = 0; i < LOOPS; i++) {
19         uint64_t dst = pattern[i % SAMPLES];
20         uint64_t idx = index[dst]++;
21         data[dst * PKT_SIZE + idx] = ~0UL;
22         if (idx == PKT_SIZE - 3) {
23             comm_send(tid, 0, dst, tid, 0, data+dst*PKT_SIZE, 0, PKT_BYTES);
24             index[dst] = 0;
25         }
26     }
27 }
28
29 /* Same as above, but receive packets. Note that not all the packets
30 * are guaranteed to be properly processed, as the calls to make_progress()
31 * end as soon as we're done injecting.
32 */
33 static void op_injrec(int tid, uint64_t *pattern, uint64_t *index, uint64_t *data)
34 {
35     for (uint64_t i = 0; i < LOOPS; i++) {
36         uint64_t dst = pattern[i % SAMPLES];
37         uint64_t idx = index[dst]++;
38         data[dst * PKT_SIZE + idx] = ~0UL;
39         if (idx == PKT_SIZE - 3) {
40             comm_send(tid, 0, dst, tid, 0, data+dst*PKT_SIZE, 0, PKT_BYTES);
41             index[dst] = 0;
42             make_progress(tid, 0);
43         }
44     }
}

```

Listing 1.1. Aggregation benchmarks

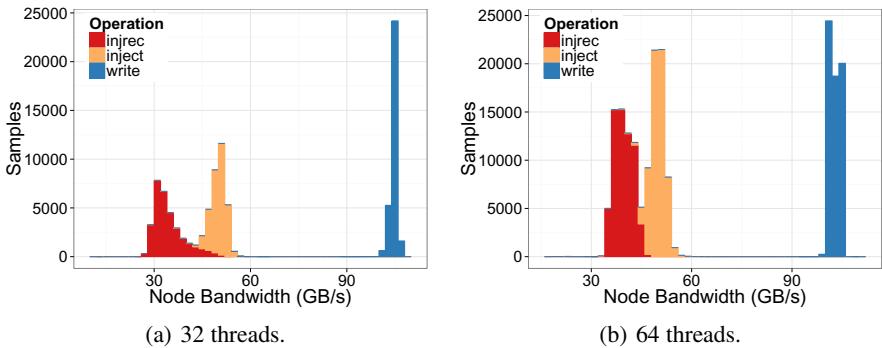


Fig. 8. Message coalescing: node bandwidth

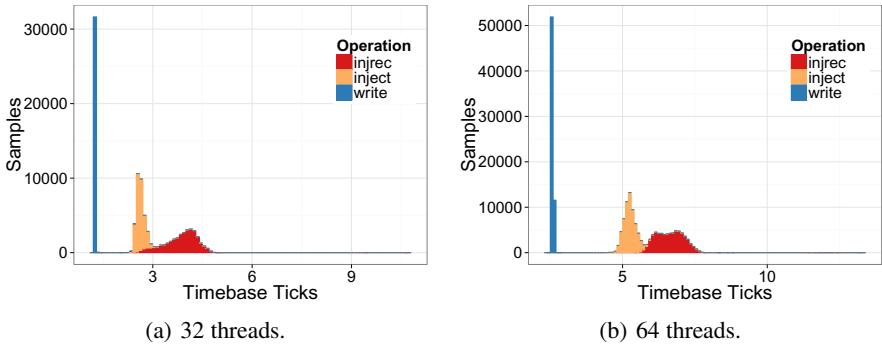


Fig. 9. Message coalescing: timing analysis

Figure 8 shows the bandwidth obtained by the three benchmarks on one supernode, using 32 and 64 threads. The `write` benchmark establishes the baseline for the purely sequential aggregation at around 100 GB/s, averaging at 102.7 GB/s on 64 threads. As soon as the interface to the network is involved, in `inject`, performance drops to an average of 49.8 GB/s. If we add an empty receive handler, in `injrec`, we see an additional performance degradation, and we get an average of 39.6 GB/s, but from the tails of the distributions we notice samples falling below the bandwidth the network would be able to sustain. With 32 threads the values are very close, averaging at, respectively, 104.4 GB/s, 49.9 GB/s, and 34.9 GB/s. The `write` benchmark is faster on 32 threads but `injrec` is faster on 64: in all likelihood hyperthreading improves efficiency by reusing some of the cycles spent polling the device registers.¹

Figure 9 shows the distributions of the time taken by each iteration (obtained averaging 8 million loops, and repeating the experiment 32 times, synchronizing the execution on each thread). The distributions mirror those obtained for the bandwidths.

¹ We experimented lowering and restoring the priorities of the hardware threads during device polling, but the results were highly dependent on the message patterns, with no clear advantage to the approach.

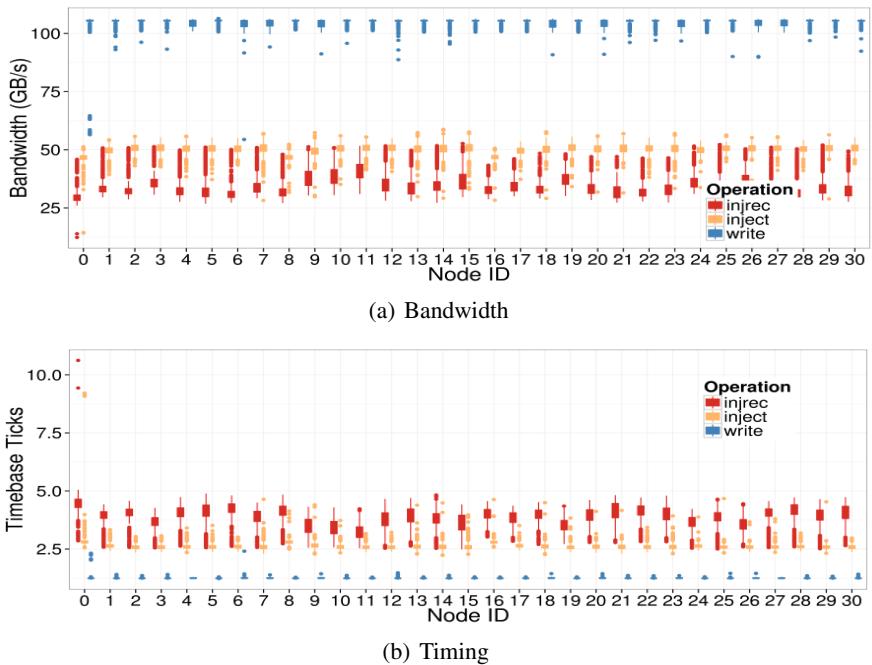


Fig. 10. Message Coalescing Distribution Analysis

Figure 10 (a) shows how the bandwidth measurements are distributed within each node when running 32 threads per node. Node 0 seems to be more exposed to noisy measurements, and presents a few outliers. The `write` performance is reasonably stable across the nodes, while both `inject` and `injrec` show more variability. Figure 10 (b) shows the corresponding distributions of the time measurements. Due to space limitation, we did not show the distribution for 64 threads per node case, which is similar to the 32 threads per node case.

After considering the simplified benchmarks, it is now time to consider the communication library in its entirety. The interface of the library is based on two calls, `send` and `flush`. `send` is not too different from `injrec`, with the only difference that supports a parametric number of distinct FIFOs/windows. `flush` takes care of counting how many messages have been sent globally, using an asynchronous *Allreduce*, and keeps polling until every node has received the messages destined to it. The all-to-all bandwidth of the library is shown in Figure 11, for a varying number of supernodes and for writes of 8 or 16 bytes per `send` call. On 16 SNs, we observed the maximum bandwidth 19.8GB/S and 22.9GB/S for 8 bytes and 16 bytes respectively. Compared to Figure 6 (b), the communication lost at least 12.6% (24.4% for 8 bytes granularity) bandwidth due to the fine-grained messages and random access to the memory and network systems.

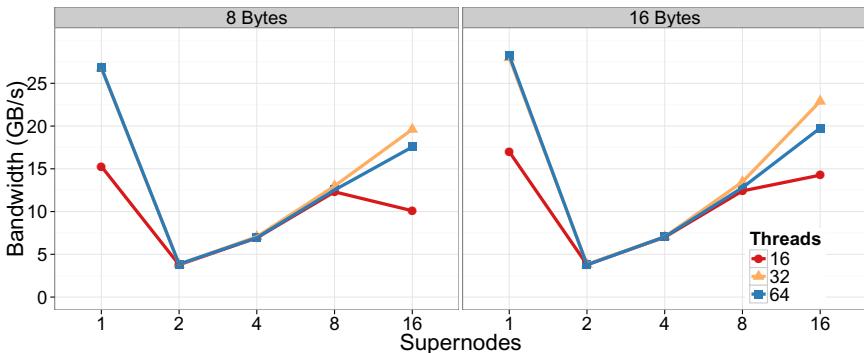


Fig. 11. Message coalescing: communication bandwidth

5 Graph Application Performance Analysis

To analyze the Graph 500 BFS kernel on P7IH, we have derived a simple implementation, which takes the advantage of our previous library for the communication. The BFS implementation is level-synchronized, i.e., it processes all the vertices at one level before moving to the next. The algorithm keeps a set of active vertices, sometimes also called a frontier, and at each iteration explores all the vertices that can be reached from the frontier in one step, storing them in a secondary queue that will become the frontier for the next iteration. Each iteration corresponds to a BFS level. It is also worth mentioning that the root is pre-generated and the central levels are always the most computationally intensive, which depends on the graph topology and the selected root.

We start from multiple metrics and benchmarks extracted from our Graph 500 implementation. In particular we tried to measure the following metrics and focused our tests on 1SN with scale factor 27 (2^{27} vertices and 2^{31} undirected edges).

- the processing rate that the system would be able to sustain if the network imposed zero overhead, as referred to **Scan**. Because it mainly consists of a linear scan of the vertex index, and a sequence of accesses to the edges that have to be processed at each level.
- The processing rate that would be achieved if the work done receiving the edges were infinitely fast, as referred to **Inject**. This allows us to take into account the effect of the network, but isolating the receive performance (both in terms of contention and cycles stolen from the injection path).
- The full performance, as referred to **Full**, including injecting and processing the received packets.

Figure 12 (a) shows the **Scan** performance of the third level (the heaviest level). In the figure we can see that the relation between the time and edges processed by each thread is close to linear (2.1×10^8 TEPS/thread with Root 2), and is only partially influenced by the size of the dataset visited (with smaller datasets being faster, as expected). Note choosing different root may result in different work load.

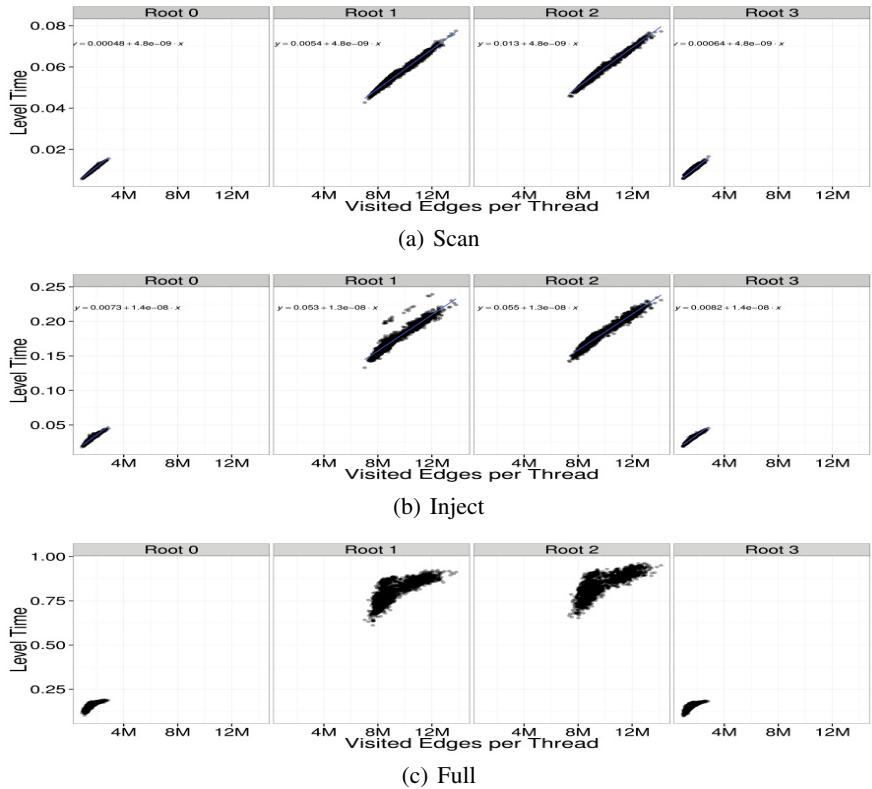


Fig. 12. Graph 500: work vs. time characterization of the third (heaviest) level with four different roots

When injecting packets into the network, but without processing them when received, the times increase as shown in Figure 12 (b), with a linear relation being somehow preserved, but with a much lower processing rate ($7.6e+7$ TEPS/thread with Root 2).

When the full receive side is considered, the linear relation is lost, as shown in Figure 12 (c), and the processing rate is further reduced ($2.0e+7$ TEPS/thread with Root 2). In particular, the processing time does not reflect the data imbalance, and the most heavily loaded threads take only a marginally higher time to finish, compared to (some of) the less loaded ones. Figure 13 (a) shows how time is spent within each level in a representative thread with Root 2. Again, we skipped the dissection for all the threads due to space limited.

The overall performance is shown in Figure 13 (b), where we examined weak scaling with 2^{27} vertices (with 2^{31} undirected edges) and 2^{28} vertices (with 2^{32} undirected edges) per node respectively. The scaling is not ideal, and on one supernode the mean processing rate is 42 GTEPS (with peaks of 50 GTEPS), while on eight supernodes it reaches 165 GTEPS (218 GTEPS peak).

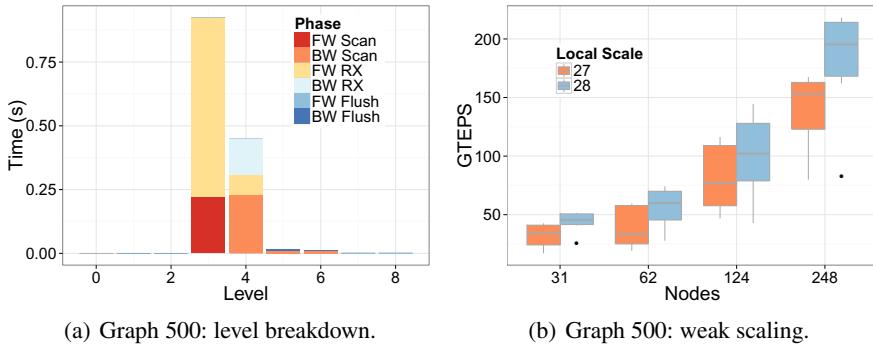


Fig. 13. Graph 500: performance

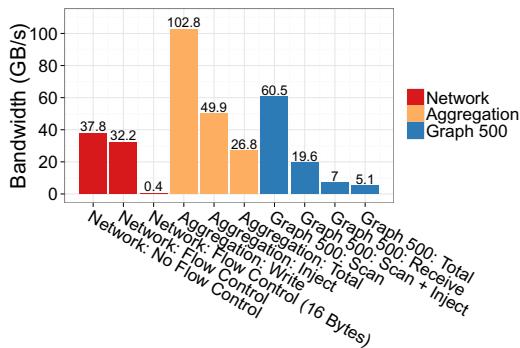


Fig. 14. Summary of Bandwidth Observed

Figure 14 summarizes the bandwidth we observed with different all-to-all scenarios on 1SN. As indicated in the figure, the P7IH can support 32.2GB/s (37.8 GB/s for Flow Control Disabled) bandwidth for plain all-to-all communication with 2032 Bytes message unit. For 16 bytes message unit, however, it only reaches 0.4 GB/s. For the fine-grained communication with irregular access pattern to memory and network system, the P7IH can deliver only 26.8 GB/s bandwidth. When it comes to the graph application, bandwidth (dividing the bytes occupied by the visited vertices by the time spent in the benchmark under consideration) is further reduced to 5.1 GB/s. The observation is that the network itself can support fast communication. Even with a high injection rate, the network is screwed because of the fine-grained message unit aggregation and irregular access pattern, which may imply that the interaction between memory subsystem and the network can not be obviated for designing architecture running data-intensive application.

6 Related Work

There are several aspects of the P7IH system have been studied previously. Kalla [8] et al. described the POWER7 chip and Arimilli [9] et al. detailed the network characteristics. However, the focuses of those studies are hardware and design decisions and our interest lies in the impact of the system to data-intensive applications. Barker [10] et al. and Kerbyson [11] et al. conducted an early drawer-level performance analysis of the P7IH system on only 8 nodes. At 16 super nodes (total 512 nodes), the system we studied is significantly larger than those earlier studies [10,11]. Tanase [12] et al. studied the non-Blocking Collective Operations on P7IH and presented a novel set of collective operations that took advantage of the P7IH hardware. Tanase [13] et al. discussed how XL Unified Parallel C (UPC) takes advantages of the hardware features available on P7IH. Rajamony [3] et al. evaluated and analyzed large-scale configurations of the P7IH on three HPC Challenge benchmarks. Different from those studies, our focus is the communication characteristics and capability of P7IH system for supporting the data-intensive graph applications.

7 Conclusions

In this paper, we detailed characterized the network performance of P7IH using a rich set of microbenchmarks. We also examined its capability of supporting fine-grained communication with irregular access pattern for graph application in detail. On top of that, we provided a vertical performance analysis of the graph algorithm, Graph500 BFS benchmark, in great details. The value of our research is that it sheds lights on the direction of future architecture design for data-intensive graph applications.

References

1. Arroyo, R.X., Harrington, R.J., Hartman, S.P., Nguyen, T.: Ibm power7 systems. *IBM J. Res. Dev.* 55(3), 220–232 (2011)
2. Rajamony, R., Arimilli, L.B., Gildea, K.: Percs: The ibm power7-ih high-performance computing system. *IBM Journal of Research and Development* 55(3), 3 (2011)
3. Rajamony, R., Stephenson, M.W., Speight, W.E.: The power 775 architecture at scale. In: *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, ICS 2013*, pp. 183–192. ACM, New York (2013)
4. Newman, M.E.J., Girvan, M.: Finding and evaluating community structure in networks. *Physical Review E* 69(2), 026113 (2004)
5. Zhang, L., Kim, Y.J., Manocha, D.: A Simple Path Non-Existence Algorithm using C-Obstacle Query. In: Proc. Intl. Workshop on the Algorithmic Foundations of Robotics (WAFR 2006), New York City (July 2006)
6. Sud, A., Andersen, E., Curtis, S., Lin, M.C., Manocha, D.: Real-time Path Planning for Virtual Agents in Dynamic Environments. In: IEEE Virtual Reality, Charlotte, NC (March 2007)
7. Chakrabarti, D., Zhan, Y., Faloutsos, C.: R-mat: A recursive model for graph mining. In: SDM (2004)
8. Kalla, R., Sinharoy, B., Starke, W.J., Floyd, M.: Power7: Ibm’s next-generation server processor. *IEEE Micro* 30(2), 7–15 (2010)

9. Arimilli, B., Arimilli, R., Chung, V., Clark, S., Denzel, W., Drerup, B., Hoefer, T., Joyner, J., Lewis, J., Li, J., Ni, N., Rajamony, R.: The percs high-performance interconnect. In: Proceedings of the 2010 18th IEEE Symposium on High Performance Interconnects, HOTI 2010, pp. 75–82. IEEE Computer Society, Washington, DC (2010)
10. Barker, K.J., Hoisie, A., Kerbyson, D.J.: An early performance analysis of power7-ih hpc systems. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2011, pp. 42:1–42:11. ACM, New York (2011)
11. Kerbyson, D.J., Barker, K.J.: Analyzing the performance bottlenecks of the power7-ih network. In: CLUSTER, pp. 244–252. IEEE (2011)
12. Tanase, G.I., Almási, G., Xue, H., Archer, C.: Composable, non-blocking collective operations on power7 ih. In: Proceedings of the 26th ACM International Conference on Supercomputing, ICS 2012, pp. 215–224. ACM, New York (2012)
13. Tanase, G., Almási, G., Tiotto, E., Alvanos, M., Ly, A., Dalton, B.: Performance analysis of the ibm xl upc on the percs architecture. Technical report, 2013. RC25360 (2013)

Sparsifying Synchronization for High-Performance Shared-Memory Sparse Triangular Solver

Jongsoo Park, Mikhail Smelyanskiy, Narayanan Sundaram, and Pradeep Dubey

Parallel Computing Lab, Intel Corporation, Santa Clara, CA, USA

Abstract. The last decade has seen rapid growth of single-chip multi-processors (CMPs), which have been leveraging Moore’s law to deliver high concurrency via increases in the number of cores and vector width. Modern CMPs execute from several hundreds to several thousands concurrent operations per second, while their memory subsystem delivers from tens to hundreds Giga-bytes per second bandwidth.

Taking advantage of these parallel resources requires highly tuned parallel implementations of key computational kernels, which form the back-bone of modern HPC. Sparse triangular solver is one such kernel and is the focus of this paper. It is widely used in several types of sparse linear solvers, and it is commonly considered challenging to parallelize and scale even on a moderate number of cores. This challenge is due to the fact that triangular solver typically has limited task-level parallelism and relies on fine-grain synchronization to exploit this parallelism, compared to data-parallel operations such as sparse matrix-vector multiplication.

This paper presents synchronization sparsification technique that significantly reduces the overhead of synchronization in sparse triangular solver and improves its scalability. We discover that a majority of task dependencies are redundant in task dependency graphs which are used to model the flow of computation in sparse triangular solver. We propose a fast and approximate sparsification algorithm, which eliminates more than 90% of these dependencies, substantially reducing synchronization overhead. As a result, on a 12-core Intel® Xeon® processor, our approach improves the performance of sparse triangular solver by 1.6x, compared to the conventional level-scheduling with barrier synchronization. This, in turn, leads to a 1.4x speedup in a pre-conditioned conjugate gradient solver.

1 Introduction

Numerical solution of sparse system of linear equations has been an indispensable tool in various areas of science and engineering for several decades. More recently, sparse solvers have gained popularity in the emerging areas of machine learning and big data analytics [15]. As a result, a new sparse solver benchmark, called HPCG (High Performance Conjugate Gradient) has been recently defined to complement HPL [21] for ranking high-performance computing systems [7].

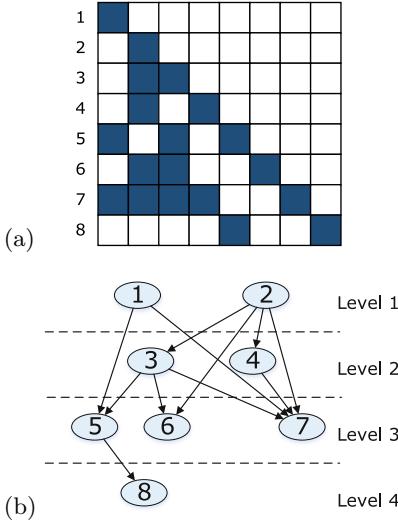


Fig. 1. (a) Non-zero pattern of a lower triangular sparse matrix and (b) its corresponding task dependency graph of forward solver with level annotations

Table 1. Evaluated sparse matrices from the University Florida collection [6] sorted by their parallelism. Parallelism is measured as (# of non-zeros)/(cumulative # of non-zeros in the rows corresponding to the longest dependency path).

	rows	nnz	/row	parallelism
1. parabolic_fem	525,825	7	75,118	
2. apache2	715,176	7	1,077	
3. thermal2	1,228,045	7	991	
4. G3_circuit	1,585,478	5	611	
5. ecology2	999,999	5	500	
6. StocF-1465	1,465,137	14	488	
7. inline_1	503,712	73	288	
8. Geo_1438	1,437,960	44	247	
9. F1	343,791	78	246	
10. bmvwra_1	148,770	72	204	
11. Emilia_923	923,136	44	176	
12. Fault_639	638,802	45	143	
13. af_shell3	504,855	35	136	
14. Hook_1498	1,498,023	41	96	
15. offshore	259,789	16	75	
16. af_3_k101	503,625	35	74	
17. BenElechi1	245,874	53	43	
18. shipsec8	114,919	58	37	
19. ship_003	121,728	66	28	
20. crankseg_2	63,838	222	15	
21. crankseg_1	52,804	201	13	

Sparse triangular solver is a key component of many sparse linear solvers and accounts for a significant fraction of their execution times. In particular, pre-conditioned conjugate gradient, using Incomplete Cholesky or Gauss-Seidel pre-conditioner, spends up to 70% of its execution time in both forward and backward solver when computing the residual of the pre-conditioned system [7, 23]. Forward and backward sweeps, used inside Multigrid Gauss-Seidel smoother, can account for up to 80% of Multigrid execution time [10].

It is hard to achieve highly scalable performance on sparse triangular solvers since their inherent parallelism is limited and fine-grain. Consider the lower triangular sparse matrix shown in Fig. 1(a). Solving the third unknown depends on the second because of the non-zero element in the third row of the second column (denoted as (3, 2)). In general, if there exists a non-zero element at (i, j) , solving the i th unknown depends on solving the j th. Based on the non-zero pattern, we can construct a task dependency graph (TDG) of computing unknowns as shown in Fig. 1(b) (note that TDGs are directed acyclic).

Table 1 lists a typical subset of scientific matrices from the University of Florida collection [6]. As Table 1 shows, the amount of parallelism, measured as the ratio between the total number of non-zeros to the number of non-zeros along the critical path in the TDG, is *limited* for most of the matrices. It is of the order of several hundreds, on average. In contrast, for sparse matrix-vector multiplication, the amount of parallelism is proportional to the number of matrix-rows, and is thus of the order of hundreds of thousands to several millions.

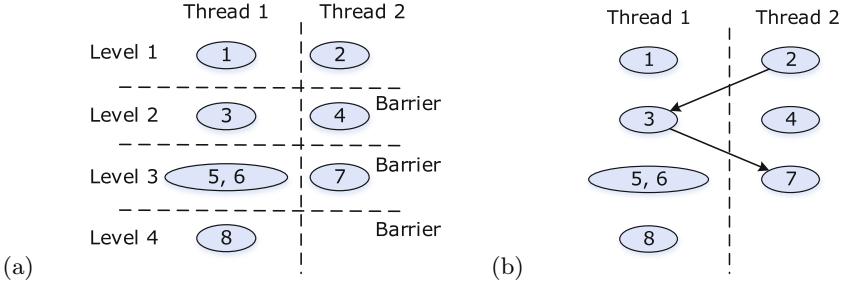


Fig. 2. Scheduling the TDG shown in Fig. 1(b) with (a) barrier synchronization and (b) point-to-point synchronization with dependency sparsification

In a TDG, computation of each task roughly amounts to the inner product of two vectors of the length equal to the number of non-zeros in the corresponding matrix row. Table 1 shows that the average number of non-zeros per row typically ranges from 7 to 222. This corresponds to only tens of hundreds of floating-point operations per row, resulting in *fine-grain* tasks within sparse solver. Considering that each core in modern processors performs tens of operations per cycle, and that core-to-core communication incurs at least tens of cycles of latency [5], synchronizing at the granularity of individual tasks can lead to prohibitive overheads.

Conventional approach to parallelize sparse triangular solvers is based on *level-scheduling with barriers* [2, 19, 24], which is illustrated in Fig. 2(a). Each level of the TDG is evenly partitioned among threads, resulting in coarse-grain *super-tasks*. The level of a task is defined by the length of the longest path from an entry node, as annotated in Fig. 1(b) [2]. When we schedule for 2 threads as in Fig. 2(a), we partition level 3 into two “super-tasks”, where the first super-task is formed out of tasks 5 and 6. Then, we synchronize after each level instead of each task, amortizing the overhead of synchronization. Still, when parallelism is limited, each barrier incurs a non-trivial amount of overhead, which increases with the number of cores.

This paper proposes a technique, called *synchronization sparsification*, which improves upon barrier-based implementation, by significantly reducing the overhead of synchronization. As Fig. 2(b) shows, in our example, we need only two point-to-point synchronizations, instead of 3 barriers. When applied to the matrices listed in Table 1, sparsification results in less than 1.6 point-to-point synchronizations per super-task, on average, which is mostly independent of the number of threads. In comparison, even the most optimized tree-based barrier synchronization requires $\log(t)$ point-to-point synchronizations per thread per level, where t is number of threads [9].

This paper makes the following contributions.

- We analyze TDGs produced from a large set of sparse matrices and observe that >90% of the edges are in fact redundant.

- We propose a fast and approximate transitive reduction algorithm for sparsifying the TDGs that quickly eliminates most of the redundant edges. Our algorithm runs orders of magnitude faster than the exact algorithm.
- Using the fast sparsification and level-scheduling with point-to-point synchronization, we implement a high-performance sparse triangular solver and demonstrate a $1.6\times$ speedup over conventional level-scheduling with barrier on a 12-core Intel® Xeon® E5-2697 v2. We further show that our optimized triangular solver accelerates the pre-conditioned conjugate gradient (PCG) algorithm by $1.4\times$ compared to the barrier-based implementation.

The rest of this paper is organized as follows. Section 2 presents our level-scheduling algorithm with sparse point-to-point synchronization, focusing on an approximate transitive edge reduction algorithm. Section 3 evaluates the performance of our high-performance sparse triangular solver and its application to PCG, comparing to a level-scheduling with barrier synchronization and the sequential MKL implementation. Section 4 reviews the related work and Section 5 concludes and discusses potential application of our approach to other directed acyclic graph scheduling problems.

2 Task Scheduling and Synchronization Sparsification

This section first presents level-scheduling with barrier synchronization—a conventional approach to parallelize sparse triangular solver. We follow with a presentation of our method, which significantly reduces synchronization overhead, compared to the barrier-based approach.

2.1 Level-Scheduling with Barrier Synchronization

The conventional level-scheduling with barrier synchronization executes TDGs one level at a time, with barrier synchronization after each level [2, 19, 24], as shown in Fig. 3(a). The level of a task is defined as the longest path length between the task and an entry node of the TDG (an entry node is a node with no parent). Since tasks which belong to the same level are independent, they can execute in parallel. In order to balance the load, we evenly partition

for each level l

```
// task $_l^t$ : super-task at level  $l$  of thread  $t$ 
solve the unknowns of task $_l^t$ 
barrier
```

(a) Level-scheduling with barrier synchronization

for each level l

```
until all the parents are done wait
solve the unknowns of task $_l^t$ 
done[task $_l^t$ ] = true
```

(b) Level-scheduling with point-to-point synchronization

Fig. 3. Pseudo code executed by thread t , solving sparse triangular system with level scheduling

(or coarsen) tasks in the same level into *super-tasks*, assigning at most one super-task to each thread. Each super-task in the same level has a similar number of non-zeros. This improves load-balance, as each thread performs a similar amount of computation and memory accesses¹. For example, assume that both tasks 5 and 6 in Fig. 1(b) have the same number of non-zeros as task 7. Then, when we schedule these three tasks on 2 threads, tasks 5 and 6 will be merged into a super-task, as shown in Fig. 2(a).

This approach is commonly used to parallelize sparse triangular solvers for the following reason. The original TDG has a fine granularity, where each task corresponds to an individual matrix row. Parallel execution of the original TDG with many fine-grain tasks would result in a considerable amount of synchronization, proportional to the number of edges in TDG, or equivalently, the number of non-zeros in the matrix. In level-scheduling with barrier, the number of synchronizations reduces to the number of levels, which is typically much smaller than the number of non-zeros.

A potential side effect of task coarsening is delaying the execution of critical paths, in particular when tasks on the critical paths are merged with those on non-critical paths. However, in level-scheduling, this delay is minimal, less than 6% for our matrices, and is more than offset by the reduction in synchronization overhead. Appendix A provides further details of quantifying the delay.

Even though level-scheduling with barrier is effective at amortizing synchronization overhead, each barrier overhead per level still accounts for a significant fraction (up to 72%) of the triangular solver time. The following section describes a method for further significantly reducing the synchronization overhead.

2.2 Level-Scheduling with Point-to-Point Synchronization

Fig. 3(b) shows pseudo-code for point-to-point (P2P) synchronization. Similar to level-scheduling with barrier, our method partitions each level into super-tasks to balance the load, while each thread works on at most one super-task per level. With each super-task, we associate a flag (denoted as `done`), which is initialized to `false`. This flag is set to `true` when the corresponds task finishes. A super-task can start when `done` flags of its parents are set to `true`. Since the flag-based synchronizations only occur between the threads executing dependent super-tasks, they are P2P, in contrast to collective synchronizations, such as barriers. Since each P2P synchronization incurs a constant overhead independent of the number of threads, this approach is more scalable than barrier synchronization, which incurs an overhead proportional to the logarithm of number of cores [9]. In addition, while barrier synchronization exposes load imbalance at each level, P2P synchronization can process multiple levels simultaneously, as long as the dependences are satisfied, thus reducing load-imbalance.

Nevertheless, if there are too many dependencies per super-task, the overhead of P2P synchronization, which processes one dependency at a time, can

¹ A more advanced approach could also account for run-time information, such as the number of cache misses.

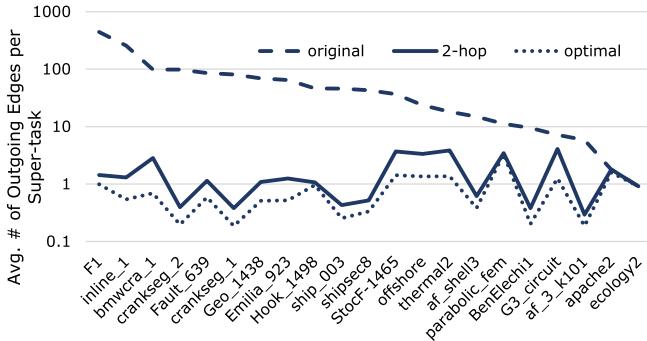


Fig. 4. The impact of transitive edge reduction. **Original**: intra-thread and duplicated edges are already removed. **Optimal**: all transitive edges are also removed. **2-hop**: transitive edges that are redundant with respect to two-hop paths are removed. Scheduled for 12 threads. The matrices are sorted by the decreasing out-degree.

exceed that of a barrier. Therefore, to take advantage of the distributed nature of P2P synchronization, one must sufficiently reduce the number of dependency edges per task. This can be accomplished by three schemes. The first eliminates *intra-thread edges* between super-tasks, statically assigned to each thread. In Fig. 1(b), tasks 2 and 4, assigned to the same thread, do not need a dependency edge, because they will naturally execute in program order. The second scheme eliminates *duplicate edges* between super-tasks. In Fig. 1(b), when tasks 5 and 6 are combined into a super-task, we need only one dependency edge from task 3 to the combined super-task. For our matrices, these two relatively straightforward schemes eliminate 49% and 8% of the edges in the original TDG, respectively. Next section describes the third, more sophisticated scheme, that enables additional large reduction in the number of edges, and, therefore, in the amount of P2P synchronization per super-task.

2.3 Synchronization Sparsification with Approximate Transitive Edge Reduction

We can further improve level-scheduling with P2P synchronization by eliminating redundant *transitive edges*. In Fig. 1(b), edge 2→6 is redundant because when task 3 finishes, and before task 6 begins, task 2 is guaranteed to be finished due to edge 2→3. In other words, edge 2→6 is a transitive edge with respect to execution path 2→3→6. The edge 1→7 is also transitive edge and therefore redundant because this order of execution will be respected due to (i) implicit schedule-imposed dependency 1→3 from the fact that both tasks execute on the same thread, and (ii) existing edge 3→7.

These transitive edges account for a surprisingly large fraction of total edges, as demonstrated by the top dotted line in Fig. 4, which shows the average number of inter-thread dependencies (outgoing edges) per super-task. As the bottom line of Fig. 4, labeled **optimal**, shows, eliminating all transitive edges results in 98%

1: $G' = G // \text{G}'$: final graph w/o transitive edges 2: for each node i in G 3: for each child k of i in G 4: for each parent j of k in G 5: if j is a successor of i in G 6: remove edge (i, k) in G'	1: $G' = G$ 2: for each node i in G 3: for each child k of i in G 4: for each parent j of k in G 5: if j is a child of i in G 6: remove edge (i, k) in G'
--	--

(a) Exact algorithm

(b) Approximate algorithm

Fig. 5. Transitive Edge Reduction Algorithms

reduction in the edges, remaining after removing intra-thread and duplicated edges by the two schemes described in Section 2.2.

We can remove all transitive edges by the algorithm due to Hsu [12], whose pseudo code is shown in Fig. 5(a). To estimate the time complexity of this algorithm, we can check whether node j is a successor of node i (i.e., j is reachable from i) in line 5 of Fig. 5(a) by running a depth-first search for each outermost loop iteration. Therefore, the time complexity of the exact transitive edge reduction algorithm is $O(mn)$. Although we are working on a coarsened TDG with typically much fewer nodes and edges than the original TDG, we assume the worst case which occurs for matrices with limited amount of parallelism. Specifically, in such cases, n and m are approximately equal to the number of rows and non-zeros in the original, non-coarsened, matrix, respectively. Therefore, $O(mn)$ overhead is too high, considering that the complexity of triangular solver is $O(m)$. Triangular solvers are often used in the context of other iterative solvers, such as PCG. In the iterative solver, the pre-processing step which removes transitive edges is done once outside of the main iterations and to be amortized over a number of iterations executed by the solver. Typically, PCG executes several hundred to several thousand of iterations, which is too few to offset the asymptotic gap of $O(n)$.

Fortunately, we can eliminate most of the redundant edges with a significantly faster approximate algorithm. Our approximate algorithm eliminates a redundant edge only if there is a two-hop path from its source to its destination. In Fig. 1(b), we eliminate a redundant transitive edge $2 \rightarrow 6$ because there exists two-hop path $2 \rightarrow 3 \rightarrow 6$. If we had an edge $2 \rightarrow 8$, it would not have been eliminated because it is redundant with respect to $2 \rightarrow 3 \rightarrow 5 \rightarrow 8$, a three-hop path. In other words, our approximate algorithm analyzes triangles, comprised of edges in the TDG, and eliminates a redundant edge from each triangle. The middle line in Fig. 4 shows that our 2-hop approach removes most ($> 98\%$) of edges, removed by the optimal algorithm: i.e., most of the edges are in fact redundant with respect to two-hop paths. This property holds consistently across all matrices.

We can remove two-hop transitive edges using the algorithm shown in Fig. 5(b). The difference from the optimal algorithm is highlighted in boldface. Since we no longer need to compute reachable nodes from each node i , this algorithm is substantially faster. The time complexity of this algorithm is $O(m \cdot E[D] + Var[D] \cdot n)$, where D is a random variable denoting the degrees of nodes in TDG; typically D

is called average number of non-zeros per row. This complexity is derived in Appendix B. This is acceptable because the average and variance of the number of non-zeros per row are usually smaller than the number of iterations of the iterative solver which calls triangular solver.

The level-scheduling methods described here statically determine the assignment and execution order of tasks. Alternatively, dynamic scheduling methods such as work stealing could change the assignment and order while executing the TDG. However, for a better load-balance, which is the main benefit of dynamic scheduling, it requires finer tasks, incurring higher synchronization overhead. In addition, static task scheduling facilitates synchronization sparsification. Specifically, while edge 2→6 in Fig. 1(b) is redundant regardless of the scheduling method including dynamic ones, 1→7 becomes redundant only when a thread is statically scheduled to execute task 1 before 3, creating a schedule-imposed edge 1→3. In dynamic scheduling, such edges cannot be removed, because the assignment and execution order of tasks are not known in advance. As a result, applying transitive-edge reduction with dynamic scheduling results in more P2P synchronizations ($\approx 1.6 \times$ for our matrices) than with static scheduling.

3 Evaluation

This section evaluates the impact of our optimizations on the performance of stand-alone triangular solver (Section 3.2), as well as the full pre-conditioned conjugate gradient (PCG) solver (Section 3.3).

3.1 Setup

We performed our experiments on a single 12-core Intel® Xeon® E5-2697 v2 socket with Ivy Bridge micro-architecture running at 2.7 GHz². It can deliver 260 GFLOPS of peak double-precision calculations, and 50 GB/s stream bandwidth. We use the latest version of Intel® Math Kernel Library (MKL) 11.1 update 1. MKL only provides an optimized sequential (non-threaded) implementations of triangular solver as well as as optimized parallel SPMVM and BLAS1 implementations, required by PCG. We use 12 OpenMP threads with `KMP_AFFINITY=sparse` since hyper threading does not provide a noticeable speedup. For level-scheduling with barrier, we use a highly optimized dissemination barrier [9]. Our barrier takes ≈ 1200 cycles, while the OpenMP barrier

² Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to <http://www.intel.com/performance>

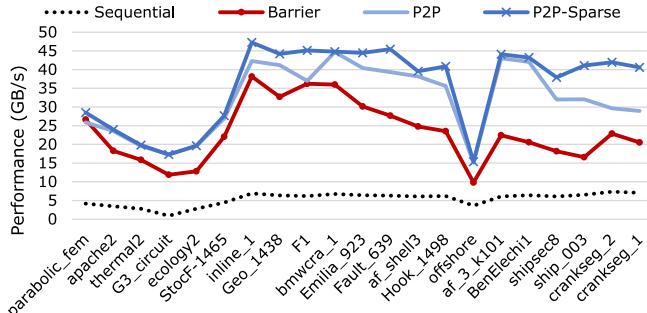


Fig. 6. Performance of sparse triangular solver with synchronization sparsification, compared with other implementations. **Barrier:** level-scheduling with barriers. **P2P:** level-scheduling with point-to-point synchronization with intra-thread and duplicate edges eliminated. **P2P-Sparse:** P2P + transitive edges eliminated. Matrices are sorted in a decreasing order of parallelism. The stream bandwidth of the evaluated processor is 50 GB/s.

takes ≈ 2200 cycles. We use Intel[®] C++ compiler version 14.0.1³. We use sparse matrices from the University of Florida collection listed in Table 1, which represent a wide range of scientific problems.

3.2 Performance of Sparse Triangular Solver in Isolation

Fig. 6 compares the performance of four triangular solver implementations. The matrices are sorted from the highest amount of parallelism (on the left) to the smallest (on the right). We report the performance in GB/s, as we typically do for sparse matrix-vector multiplication, because solving sparse triangular systems is bound by the achievable bandwidth when corresponding sparse matrix has sufficient amount of parallelism and thus low overhead of synchronization. Since our matrices are stored in the compressed row storage (CRS) format, the performance is thus computed by dividing $12m$ bytes by the execution time, where m is the number of non-zero elements, and 12 is the number of bytes per non-zero element (8-byte value and 4-byte index). Here, we conservatively ignore extra memory traffic that may come from the accesses to left and right hand-side vectors. As Fig. 6 shows, level-scheduling with sparse point-to-point

³ Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804.

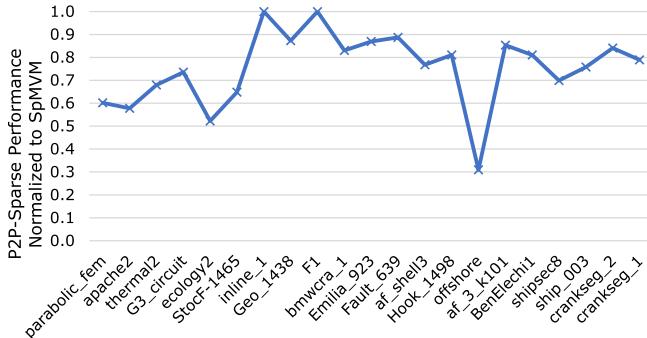


Fig. 7. The performance of **P2P-Sparse** normalized to that of **SpMVM**. Their gap can be related to the relative synchronization overhead.

synchronization (**P2P-Sparse**) is on average $1.6 \times$ faster than the one with barrier (**Barrier**), and the performance advantage of **P2P-Sparse** is in general wider for matrices with limited parallelism. In other words, **P2P-Sparse** is successful at sustaining comparable levels of performance for matrices with different amounts of parallelism.

Point-to-point implementation with intra-thread and duplicate edge elimination but without transitive edge elimination (P2P) is on average 11% slower than **P2P-Sparse**, in particular for matrices with limited parallelism, demonstrating the importance of sparsification. Overall, the sparsification reduces the number of outgoing edges per super-task to a small number, which is similar for a wide range of matrices, thus resulting in comparable levels of performance for these matrices.

The performance of sparse triangular solver is bound by that of **SpMVM**. While they involve the same number of floating point operations, **SpMVM** is embarrassingly parallel, in contrast to the sparse solver with limited parallelism. Fig. 7 plots the performance of **P2P-Sparse** normalized to that of **SpMVM**⁴. Our triangular solver achieves on average 74% of the **SpMVM** performance, and successfully sustains $>70\%$ of the **SpMVM** performance for matrices with as little as ten-fold amount of parallelism. A large fraction of the performance gap is from synchronization overhead. For example, **offshore** has a low relative performance because of frequent synchronization: each super-task is very fine-grain because of its smaller parallelism (more levels) and fewer non-zeros per row. We expect that the synchronization overhead will play an even more significant role as scaling the number of cores and the memory bandwidth. Suppose a new processor with $a \times$ cores and $b \times$ stream bandwidth per core. In a strong scaling, each task will access $a \times$ fewer bytes, and they will be transferred in a $b \times$ faster rate. To keep

⁴ In Figures 6 and 7, even though we are reporting with **SpMVM** and triangular solver performances separately, and exclude the pre-processing time, we measure their performance within PCG iterations. This is necessary to maintain a realistic working set in the last-level caches.

```

// pre-processing
1: find_levels( $A$ )           // not for MKL
2:  $TDG = \text{coarsen}(A)$  // for P2P and P2P_Sparse
   // Form super tasks, eliminate intra-thread and duplicated edges.
3: sparsify( $TDG$ )          // for P2P_Sparse
4:  $L = \text{incomplete\_Cholesky}(A)$ 
// main loop
5: while not converged
6:   1 SPMVM with  $A$ 
7:   1 forward/backward triangular solve with  $L/L^T$ 
8:   a few BLAS1 routines

```

Fig. 8. Pseudo code of conjugate gradient with Incomplete Cholesky pre-conditioner

the relative synchronization overhead the same, we need $ab \times$ faster inter-core communication.

3.3 Pre-conditioned Conjugate Gradient

Conjugate gradient method (CG) is the algorithm of choice for iteratively solving systems of linear equations when the accompanying matrix is symmetric and positive-definite [11]. CG is often used together with a pre-conditioner to accelerate its convergence. A commonly used pre-conditioner is Incomplete Cholesky factorization, whose application involves solving triangular systems of equations [18].

We implement a pre-conditioned conjugate gradient solver with Incomplete Cholesky pre-conditioner (ICCG). Fig. 8 shows a schematic pseudo-code of ICCG solver which highlights the key operations. Steps 1, 2 and 3 perform matrix pre-processing. They are required by our optimized implementation of forward and backward solvers, called in step 7 and are accounted for in the run-time of ICCG. We use the unit-vector as the right-hand side input, and the zero-vector as the initial approximation to the solution. We iterate until the relative residual reaches below 10^{-7} .

Fig. 9 breaks down the ICCG execution time using different versions of triangular solver. The execution time is normalized with respect to the total ICCG time using MKL. As mentioned in Section 3.1, MKL uses an optimized sequential triangular solver and Incomplete Cholesky factorization. Since triangular solver is not parallelized in MKL, it accounts for a large fraction of the total time. Using our optimized parallel P2P_Sparse implementation significantly reduces its execution time and makes it comparable to that of SPMVM. Note that the pre-processing time for finding levels, coarsening tasks, and sparsifying synchronization is very small (<1.5%).

Due to faster triangular solver, P2P-Sparse speedups ICCG by $1.4 \times$, compared to the barrier implementation. Incomplete Cholesky factorization time (**Factor**) is insignificant except for **crankseg** matrices that converges in a few iterations, thus exposing its overhead. Since Incomplete Cholesky uses the same TDG as

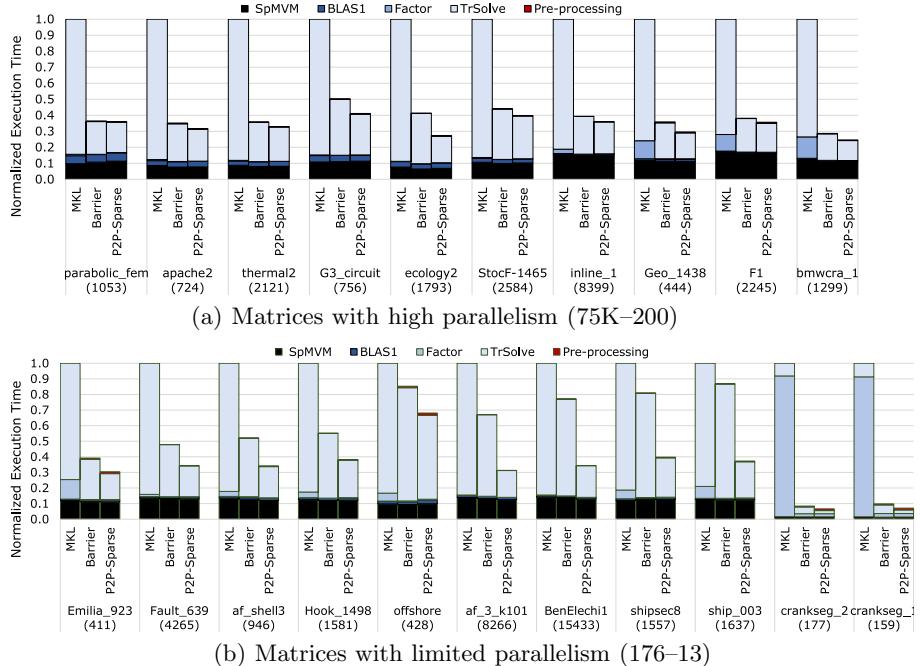


Fig. 9. Execution time breakdowns of conjugate gradient using the Incomplete Cholesky pre-conditioner. The parentheses below matrix names contain the number of ICCG iterations.

triangular solver, it benefits from the same parallelization strategies. In comparison to triangular solver, **Barrier** and **P2P-Sparse** however result in a small performance difference in Incomplete Cholesky factorization because tasks are not as fine as in triangular solver.

Fig. 10 quantifies the pre-processing overhead of our triangular solver. Pre-processing overhead is comparable to just a few ICCG iterations, which explains its small fraction of overall execution time, as shown in Fig. 9. This is the result of our highly optimized implementation. A modified breadth-first search is used when finding levels (`find_levels`), which is based on the implementation by Chhugani et al. [4]. In `coarsen`, most of time is spent sorting adjacency lists of super-tasks to eliminate duplicated edges. This sorting also facilitates quickly identifying 2-hop transitive edges in `sparsify`.

Based on the pre-processing time, we compute the break-even number of ICCG iterations when additional pre-processing overhead for **P2P-Sparse** is amortized due to coarsening and sparsification and thus **P2P-Sparse** becomes faster than **Barrier**. On average, across all of our evaluated matrices, this happens when ICCG runs for 15 iterations.

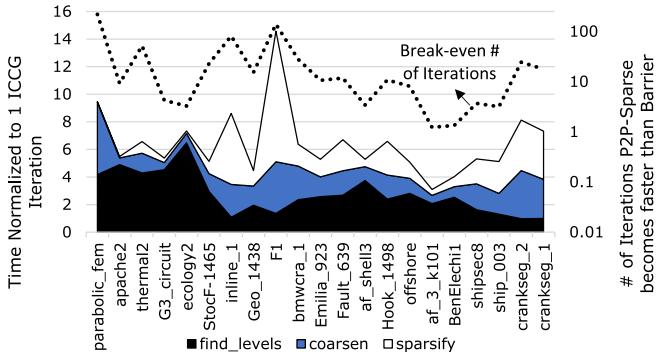


Fig. 10. Pre-processing time breakdowns (stacked areas), and the break-even point number of ICCG iterations when P2P-Sparse becomes faster than **Barrier** (the dotted line). **find_levels**, **coarsen**, and **sparsify** correspond to steps 1–3 in the ICCG pseudo-code (Fig. 8).

4 Related Work

Due to its parallelization challenges, sparse triangular solvers have been a limiting performance factor in various numerical algorithms. To address this limitation, there have been several efforts on improving its scalability. Anderson and Saad [2] and Saltz [24] present level scheduling method, which is also adopted by Naumov for Nvidia GPUs [19]. Rothberg and Gupta [23] show that dynamic scheduling algorithms perform worse because of the synchronization overhead when exploiting fine-grain parallelism. Wolf et al. [27] evaluate their level-scheduling with barrier implementation on an 8-core Intel® Nehalem and a 12-core AMD Istanbul system. For a matrix with 15.1 parallelism (**bcsstk32**), they report slowdowns in both systems when the maximum number of threads are used. For the same matrix, we observe that **Barrier** achieves $1.2\times$ and P2P-Sparse achieves $3.9\times$ parallelization speedups with 12 threads. Although they are not directly comparable due to different hardware, it nevertheless highlights the benefits of our approach. Mayer [17] presents a 2D decomposition scheme that can potentially provide more parallelism than the 1D decomposition where a row is the minimum unit of scheduling. Although he reports modest speedups, his method can be useful for larger and denser matrices where 2D decomposition will not result in too many fine-grain tasks; we expect that our synchronization sparsification will be also useful there.

While this paper *exploits* available parallelism by minimizing the synchronization overhead, alternatively, one can further *increase* the amount of parallelism by reordering the matrix. Multi-color reordering [22] identifies the rows that do not have any non-zero elements in the same column so that they can be solved in parallel. The reordering scheme can be formulated as the graph coloring problem on the undirected adjacency graph induced from the sparse matrix. Since multi-coloring breaks transitive dependency edges (e.g., we can reorder *a* and *c* even

with $a \rightarrow b \rightarrow c$ as long as a and c are not directly connected), it typically finds orders of magnitude more parallelism. However, multi-color reordering often degrades the convergence rate of iterative solvers, partially reducing the benefits of using pre-conditioners [14]. It also often degrades the locality of accessing the input vector, and exposes additional pre-processing overhead due to graph coloring. Nevertheless, multi-color reordering can be an indispensable tool for massively parallel processors such as GPUs and Intel® Xeon Phi™ coprocessors, and will complement the sparsification approach developed in this paper. Although not presented in this paper, due to its focus on matrices with limited parallelism, we did see performance improvements from the synchronization sparsification even in the multi-color reordered matrices.

There has been other work that compares global synchronization schemes with local (or P2P) ones. Specifically, our P2P scheme resembles the self-executing scheduling that is distinguished from the level-scheduling with barriers by Saltz et al. [25]. Finally, for a different application domain, Park and Dally [20] present a scheduling algorithm with P2P synchronization, which is more efficient than a barrier-based algorithm. They also apply task merging to amortize synchronization overhead and eliminate unnecessary inter-task synchronizations. Scheduling with directed acyclic graph also as been applied to dense linear algebra operations (SuperMatrix [3] and PLASMA [1]) and to sparse linear algebra operations (Kim and Eijkhout [16]). They partition matrices into small blocks and express operations as task dependency graphs where each task corresponds to operations on the tiles (This approach is called algorithm-by-blocks in comparison to blocked algorithms [3]). However, the granularity of task dependency graphs evaluated in this paper is considerably finer, where task coarsening and redundant dependency elimination is critical for the performance.

Another challenge of triangular solver is its memory access pattern with less locality compared to spmvm. We reorganize matrices so that the data are accessed sequentially as done by Smith and Zhang [26].

5 Conclusion

Our synchronization sparsification technique achieves a $1.6\times$ speedup in sparse triangular solver compared to the conventional level-scheduling with barriers. Our fast approximate transitive reduction algorithm results in minimal pre-processing overhead of sparsification, thus leading to a $1.4\times$ speedup in a preconditioned conjugated gradient solver.

Our technique is not limited to triangular solver. Similar task dependency graphs are constructed for incomplete-Cholesky/LU factorization and the smoothing step in multi-grid method. Our approach may also be useful in general directed acyclic graph scheduling problems outside sparse matrix operations.

A Delay of Critical Paths in Level-Scheduling

We can quantify the delay by comparing a coarsened schedule with a fine-grain one optimized for critical paths. Since finding a schedule that minimizes the execution time

of critical paths is NP-hard, Highest Level First with Estimated Time (HLFET) [13] heuristic algorithm is commonly used. We run HLFET on the original fine-grain task graph and obtain the scheduled completion time of critical paths, assuming each task execution time is proportional to its number of non-zeros and ignoring synchronization overhead. When we coarsen task graphs with level-scheduling, its completion time is delayed by on average less than 6% compared to that of HLFET, for matrices listed in Table 1.

It is an open problem to further minimize the delay from task coarsening. If not careful, merging tasks can delay the critical path significantly. Suppose edges $a_i \rightarrow b_i$ for many a_i and b_i s. Merging a_{i+1} s with b_i s creates an arbitrary long dependency chain. Merging tasks can even result in a deadlock [20]. In Fig. 1(b), merging 2 and 7 introduces a cycle $(2, 7) \rightarrow 4 \rightarrow (2, 7)$, causing a deadlock. Level-scheduling is obviously deadlock free, and one can show that the maximum delay in critical path is $2 \times$ (a similar argument to the $2 \times$ bound of list scheduling [8]).

B Time Complexity of 2-Hop Transitive Edge Reduction

Suppose that the adjacency and inverse adjacency list of each node are sorted. In lines 4–6 of Fig. 5(b), for each child k of i , we can check if there is a common element between i 's children and k 's parents in $O(\max(d_{out}(i), d_{in}(k)))$, where $d_{in/out}(i)$ denotes the in/out degree of node i . Therefore, the time complexity of this algorithm is

$$\begin{aligned} & \sum_i \sum_{k \in \text{Children}(i)} O(\max(d_{out}(i), d_{in}(k))) \\ &= O\left(\sum_i d(i)^2\right), \quad (d(i) = d_{in}(i) + d_{out}(i)) \\ &= O((E[D]^2 + Var[D])n), \quad (D : \text{random variable for } d(i)s, E[D] = \frac{m}{n}) \\ &= O(m \cdot E[D] + Var[D] \cdot n). \end{aligned}$$

Acknowledgements. The authors would like to thank anonymous reviewers for suggesting an alternative point-to-point synchronization scheme and related work to reference. We thank David S. Scott, Nadathur Satish, and Alexander A. Kalinkin for discussion during the initial stage of our project. We also thank Michael A. Heroux for his insights related to the implementation and performance of conjugate gradient, and Kiran Pamnany for sharing his dissemination barrier implementation.

References

- [1] Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P., Tomov, S.: Numerical linear algebra on emerging architectures: the PLASMA and MAGMA projects. Journal of Physics: Conference Series 180 (2009)

- [2] Anderson, E., Saad, Y.: Solving Sparse Triangular Linear Systems on Parallel Computers. *International Journal of High Speed Computing* 1(1) (1989)
- [3] Chan, E., Quintana-Orti, E.S., Quintana-Orti, G., van de Geijn, R.: SuperMatrix Out-of-Order Scheduling of Matrix Operations for SMP and Multi-Core Architectures. In: *Symposium on Parallelism in Algorithms and Architectures (SPAA)* (2007)
- [4] Chhugani, J., Satish, N., Kim, C., Sewall, J., Dubey, P.: Fast and Efficient Graph Traversal Algorithm for CPUs: Maximizing Single-Node Efficiency. In: *International Symposium on Parallel and Distributed Processing (IPDPS)* (2012)
- [5] Molka, R.S.D., Hackenberg, D., Müller, M.S.: Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System. In: *International Conference on Parallel Architectures and Compilation Techniques (PACT)* (2009)
- [6] Davis, T.A., Hu, Y.: The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software* 15(1) (2011), <http://www.cise.ufl.edu/research/sparse/matrices>
- [7] Dongarra, J., Heroux, M.A.: Toward a New Metric for Ranking High Performance Computing Systems. Technical Report 4744, Sandia National Laboratories (2013)
- [8] Graham, R.L.: Bounds on Multiprocessing Timing Anomalies. *SIAM Journal on Applied Mathematics* 17(2) (1969)
- [9] Hensgen, D., Finkel, R., Manber, U.: Two Algorithms for Barrier Synchronization. *International Journal of Parallel Programming* 17(1) (1988)
- [10] Henson, V.E., Yang, U.M.: Boomeramg: a parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics* 41, 155–177 (2000)
- [11] Hestenes, M.R., Stiefel, E.: Methods of Conjugate Gradients for Solving Linear Systems. *Journal of Research of the National Bureau of Standards* 49(6) (1952)
- [12] Hsu, H.T.: An Algorithm for Finding a Minimal Equivalent Graph of a Digraph. *Journal of the ACM (JACM)* 22(1) (1975)
- [13] Hu, T.C.: Parallel Sequencing and Assembly Line Problems. *Operations Research* 19(6) (1961)
- [14] Iwashita, T., Nakashima, H., Takahashi, Y.: Algebraic Block Multi-Color Ordering Method for Parallel Multi-Threaded Sparse Triangular Solver in ICCG Method. In: *International Symposium on Parallel and Distributed Processing (IPDPS)* (2012)
- [15] Kepner, J., Gilbert, J.: *Graph Algorithms in the Language of Linear Algebra*. Society for Industrial & Applied Mathematics (2011)
- [16] Kim, K., Eijkhout, V.: A Parallel Sparse Direct Solver via Hierarchical DAG Scheduling. Technical Report 5, The Texas Advanced Computing Center (2012)
- [17] Mayer, J.: Parallel algorithms for solving linear systems with sparse triangular matrices. *Computing* 86(4) (2009)
- [18] Meijerink, J.A., van der Vorst, H.A.: An Iterative Solution Method for Linear Systems of Which the Coefficient Matrix is a Symmetric M-Matrix. *Mathematics of Computation* 31(137) (1977)
- [19] Naumov, M.: Parallel Solution of Sparse Triangular Linear Systems in the Preconditioned Iterative Methods on the GPU. Technical Report 001, NVIDIA Corporation (2011)
- [20] Park, J., Dally, W.J.: Buffer-space Efficient and Deadlock-free Scheduling of Stream Applications on Multi-core Architectures. In: *Symposium on Parallelism in Algorithms and Architectures (SPAA)* (2010)
- [21] Petitet, A., Whaley, R.C., Dongarra, J., Cleary, A.: HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers, <http://www.netlib.org/benchmark/hpl/>

- [22] Poole, E.L., Ortega, J.M.: Multicolor ICCG Methods for Vector Computers. *SIAM Journal on Numerical Analysis* 24(6) (1987)
- [23] Rothberg, E., Gupta, A.: Parallel ICCG on a Hierarchical Memory Multiprocessor - Addressing the Triangular Solve Bottleneck. *Parallel Computing* 18(7) (1992)
- [24] Saltz, J.H.: Aggregation Methods for Solving Sparse Triangular Systems on Multiprocessors. *SIAM Journal of Scientific and Statistical Computing* 11(1) (1990)
- [25] Saltz, J.H., Mirchandaney, R., Baxter, D.: Run-Time Parallelization and Scheduling of Loops. In: *Symposium on Parallelism in Algorithms and Architectures (SPAA)* (1989)
- [26] Smith, B., Zhang, H.: Sparse triangular solves for ILU revisited: Data layout crucial to better performance. *International Journal of High Performance Computing Applications* 25(4), 386–391 (2011)
- [27] Wolf, M.M., Heroux, M.A., Boman, E.G.: Factors Impacting Performance of Multithreaded Sparse Triangular Solve. In: Palma, J.M.L.M., Daydé, M., Marques, O., Lopes, J.C. (eds.) *VECPAR 2010. LNCS*, vol. 6449, pp. 32–44. Springer, Heidelberg (2011)

Scalability and Parallel Execution of OmpSs-OpenCL Tasks on Heterogeneous CPU-GPU Environment

Vinoth Krishnan Elangovan^{1,2}, Rosa. M. Badia^{1,3}, and Eduard Ayguadé^{1,2}

¹ Barcelona Supercomputing Center, Spain

² Universitat Politècnica de Catalunya, Spain

³ Artificial Intelligence Research Institute (IIIA),
Spanish National Research Council (CSIC), Spain

Abstract. With heterogeneous computing becoming mainstream, researchers and software vendors have been trying to exploit the best of the underlying architectures like GPUs or CPUs to enhance performance. Parallel programming models play a crucial role in achieving this enhancement. One such model is OpenCL, a parallel computing API for cross platform computations targeting heterogeneous architectures. However, OpenCL is a low-level programming language, therefore it can be time consuming to directly develop OpenCL code. To address this shortcoming, OpenCL has been integrated with OmpSs, a task-based programming model to provide abstraction to the user thereby reducing programmer effort. OmpSs-OpenCL programming model deals with a single OpenCL device either a CPU or a GPU. In this paper, we upgrade OmpSs-OpenCL programming model by supporting parallel execution of tasks across multiple CPU-GPU heterogeneous platforms. We discuss the design of the programming model along with its asynchronous runtime system. We investigated scalability of four OmpSs-OpenCL benchmarks across 4 GPUs gaining speedup of up to 4x. Further, in order to achieve effective utilization of the computing resources, we present static and work-stealing scheduling techniques. We show results of parallel execution of applications using OmpSs-OpenCL model and use heterogeneous workloads to evaluate our scheduling techniques on a heterogeneous CPU-GPU platform.

1 Introduction

In the past decade, in order to deliver performance improvements, microprocessors vendors chose multi-core design paradigm to overcome memory, power and ILP walls. Today, multi-core CPUs package multiple homogeneous cores on a single die to increase data parallel computations. On the other hand, GPUs, with immense data parallel processing capability, are being exploited for general purpose computing (GPGPU) which traditionally handled graphics computations. Offering massive data parallel computing power, GPUs have become the focal point of today's High Performance Computing (HPC). Considering the recent progress of major chip manufacturers it is very clear that

in the future, laptops to HPC systems will consist of heterogeneous computing devices (CPU/GPU/DSP/FPGA). Thus presenting us with a hybrid/heterogeneous computing environment. This poses software developers with a significant challenge of best utilizing the underlying hardware. To harness this immense computing power for HPC, hardware vendors have built platform specific programming models like CUDA[16]. However, these models are quite demanding and involve significant software development time. Furthermore, these models suffer from portability issues, thereby pushing developers to rewrite code for different platforms from scratch. This makes heterogeneous programming quite tedious. In order to address this *state of the art* parallel programming, Khronos group[11][15] came up with OpenCL, an open source, platform independent, parallel computing API offering code portability. OpenCL solves the problem of using heterogeneous computing environment with a single programming model, but requiring significant programming effort for effective use. To address this issue, our previous work [8] proposes OmpSs-OpenCL programming model. It focuses on integrating OpenCL with OmpSs and discusses in detail the abstraction of OpenCL features and the semantics of OmpSs-OpenCL model. However, the model provides support either for a single GPU or a CPU. In this paper, we enhance OmpSs-OpenCL programming model to provide support for parallel execution of tasks in a CPU-GPU heterogeneous environment. Figure 1 depicts the architectural overview of the OmpSs-OpenCL programming model. The model includes the Mercurium compiler and the Nanos runtime system. The compiler does a source-to-source transformation of the user code written using OmpSs-OpenCL. During compilation the source code is embedded with appropriate Nanos runtime calls responsible for incorporating automatic data transfers and OpenCL execution. The runtime carries out task creation, task dependency analysis and scheduling. The Nanos software cache helps in tracking data locality of OmpSs-OpenCL tasks. This programming model supports all OpenCL compliant devices under a single operating system image, for instance, a system with 2 CPUs and 4 discrete GPUs can be programmed using OmpSs-OpenCL model with all 6 devices operating simultaneously.

With OmpSs-OpenCL supporting parallel execution of tasks across heterogeneous devices, there arises an opportunity to schedule these tasks efficiently across devices. In this paper, we discuss two scheduling methodologies namely static and work-stealing strategy. The static scheduling follows user specification of the target device for execution whereas the work-stealing essentially schedules tasks to the device that is devoid of work. The major contributions of this paper are the following:

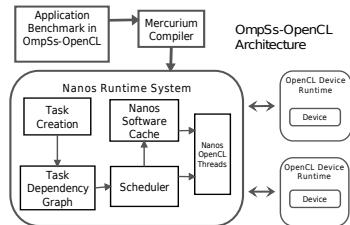


Fig. 1. OmpSs-OpenCL Programming Model - Architecture

- We enhance OmpSs-OpenCL model to support heterogeneous/hybrid environment supporting parallel execution of tasks.
- We present the design of the programming model for CPU-GPU systems and discuss its supporting runtime system.
- We discuss scalability and pre-fetching of OmpSs-OpenCL tasks and evaluate proposed scheduling strategies using heterogeneous workloads.

This paper is divided into seven sections. In the subsequent section, we describe the design of OmpSs-OpenCL programming model. In section 3, we discuss the details of Mercurium compiler and Nanos runtime system for heterogeneous environments. Following that, in section 4, we present the scalability and pre-fetching of OmpSs-OpenCL tasks. In section 5, we present static and work-stealing scheduling along with their evaluation using heterogeneous workloads. We give an overview of the related work in section 6 and finally conclude with promising future extensions to this work in section 7.

2 OmpSs-OpenCL Model

OmpSs[7] is based on the OpenMP programming model with modifications to its execution model. It is primarily a task-based programming model focusing on abstraction of details to the user thereby making the programmer to write code in sequential flow with annotated pragmas for tasks(parallel regions). It uses a thread-pool execution model, where a master thread that starts the runtime and several other worker threads cooperate towards executing the tasks. Listing1.1 shows the directives supported by OmpSs-OpenCL model. These directives are used to annotate function declarations or its definitions. Each function annotated with task directives is considered an OmpSs-OpenCL task. The data environment of the task is obtained from its arguments. These arguments are specified with their directionality *input*, *output* and *inout* and its computing size. Using this information, dependencies across tasks are determined using StarSs dependency model[3]. Furthermore, *target device* clause is used to express heterogeneity, which can be *clcpu* and *clgpu*. *clcpu* undergoes OpenCL CPU execution and *clgpu* OpenCL GPU execution.

```

1 #pragma omp target device [clauses] NDRange(Parameters)
    [copy_type]
2 clauses : ([clcpu][clgpu])
3 Parameters : Dimensions, GlobalGrid, LocalWorkGroupSize
4 copy-type : copy_in, copy_out, copy_inout, copy_deps
5 #pragma omp task [directionality]
6 Directionality
7 1. input ([list of parameters])
8 2. output ([list of parameters])
9 3. inout ([list of parameters])
10 #pragma omp taskwait

```

Listing 1.1. Directives supported by OmpSs-OpenCL

The *copy_in*, *copy_out* and *copy_inout* clauses are used to specify where the data have to be available, produced and both. *copy_deps* clause is used to specify that if the task has any dependence clauses, then they will also have copy semantics. The task-wait construct (Listing1.1-line 10) can be used to introduce a barrier after parallel code. Along with these directives, NDRange will accept OpenCL execution configuration for the particular task/kernel. These parameters essentially represent the OpenCL grid dimensionality[11], *NDRange* Global Grid and the *LocalWorkGroupSize*[11]. Moreover, in OmpSs-OpenCL, the task definition is essentially an OpenCL kernel written according to OpenCL C99 standard by the user. The task/kernel code can be defined in the same file of the source or in a separate .cl file. The following sample gives a comprehensive understanding of how to use the OmpSs-OpenCL directives(Listing 1.2).

```

1 #pragma omp target device (clccpu) ndrange(1,0,size,512)
2   copy_deps
3
4 #pragma omp task in ([size]a) out([size]c)
5 void copy_task(double *a, double *c, int size);
6
7 #pragma omp target device (clgpu) ndrange(1,0,size,512)
8   copy_deps
9 #pragma omp task in ([size]c) out([size]b)
10 void scale_task (double *c, double *b, int size);
11
12 #pragma omp target device (clgpu) ndrange(1,0,size,512)
13   copy_deps
14 #pragma omp task in ([size]a,[size]b) out([size]c)
15 void add_task (double *a, double *b, double *c, int
16   size);
17
18 int main(int argc, char** argv)
19 {
20   copy_task(a,c,size);
21   scale_task (c,b,size);
22   add_task(a, b,c,size);
23   triad_task (b, c, a,size);
24 }
```

Listing 1.2. OmpSs-OpenCL Example Program

Listing 1.2 is an example code in OmpSs-OpenCL for heterogeneous environment. As shown, the directives are used to specify on which device the task should execute (*clccpu* or *clgpu*). The directive *task* mentions the required data by the task in lines 2,6,10 and 14. Also, *target device* for the task is mentioned

in lines 1,5,9 and 13. When the tasks are invoked, the runtime checks for data dependencies between them, if independent, they can be executed in parallel in the CPU-GPU environment. Listing-1.2 example program is a stream application [14]. The definition of the tasks is essentially a OpenCL kernel. The kernel code for *triad_task* is shown in listing 1.3.

```

1 #pragma OPENCL EXTENSION cl_khr_fp64 : enable
2 __kernel void triad_task ( __global double *a, __global
3                           double *b, __global double *c, __const double
4                           scalar, __const int size)
5 {
6     int j=get_global_id(0);
7     a[j] = b[j]+scalar*c[j];
8 }
```

Listing 1.3. OmpSs-OpenCL Task/Kernel

As per specification, *copy_task* (line 19) is the only task in the benchmark undergoing OpenCL CPU execution and the remaining are executed in the OpenCL GPU. We can apprehend from this example that the user can avoid tedious OpenCL API calls and also realize parallel execution across different devices, leaving users to write the task/kernel code alone. The next section gives a detailed description of the Nanos runtime.

3 OmpSs-OpenCL Execution Model

In this section, we discuss the implementation details of OmpSs-OpenCL programming model for hybrid environments. The model embodies Mercurium compiler and the Nanos runtime system.

3.1 Mercurium Compiler

Mercurium compiler [4] is a source-to-source compilation infrastructure supporting C and C++. It checks syntax and semantic errors for annotated pragmas in the program and parses the OmpSs-OpenCL source generating the associated runtime calls for the parallel regions/tasks to be executed. These runtime calls embodies information about the directionality of the data transfers in order to deduce input/output dependencies among all tasks in the source. With inclusion of *clcpu* or *clgpu* as *target device*, necessary changes to the compiler have been carried out to generate their respective OpenCL-Nanos runtime calls. When Mercurium encounters a task declaration or invocation which is targeted for *clcpu* or *clgpu* it generates corresponding task creation runtime call with the respective .cl file as a parameter to it. The task/kernel code written according to OpenCL standard is left untouched by the compiler as it undergoes target device specific runtime compilation. Mercurium is only responsible to generate appropriate calls to the runtime and check the correctness of user code. The back-end of the model involving Nanos runtime does the significant portion of the work in order to experience parallel execution of tasks across multiple OpenCL devices.

3.2 Nanos Runtime Environment

The Nanos runtime library is the backbone of OmpSs-OpenCL programming model. It is a task-based runtime system with asynchronous execution flow. It creates an acyclic task dependency graph based on the *task* directive information. The runtime carries out the services including task creation, task dependency graph generation, data transfers, synchronization between tasks and execution of tasks. The Nanos master thread is responsible for most of these services whereas the worker threads are accountable for task execution. Each Nanos worker thread corresponds to a device in the heterogeneous environment and work in parallel. In the following subsections we discuss the enhancement implemented in the runtime to support hybrid OpenCL environments. Figure 2 presents OmpSs-OpenCL model diagrammatically.

Support for Heterogeneity. Nanos runtime essentially invokes a master thread which in turn controls the worker threads. These worker threads (*Nanos-arch* thread) implements the functionalities for different architectures. For OmpSs-OpenCL model the Nanos-OpenCL worker thread implements OpenCL execution for a CPU or GPU device[8]. However, for hybrid environments the equation changes. Although, OpenCL API is open-source but the implementation of its runtime varies with every vendor (eg., Nvidia, Intel, AMD, ARM ..). Hence, not all devices can use the same OpenCL package. We need to have vendor-specific OpenCL package in order to use different devices. To tackle this issue we implement two different OpenCL-Nanos architectures (Nanos-OpenCL-CPU thread and Nanos-OpenCL-GPU thread), to realize both CPU and GPU OpenCL behavior respectively. The Nanos-*arch*-OpenCL Thread is responsible for compilation, argument setting and execution of the task/kernel. With CPU and GPU OpenCL architecture module, OmpSs-OpenCL supports all OpenCL compliant CPU and GPU systems. Figure 2 shows the Nanos-OpenCL-*device* thread model linking with the corresponding OpenCL package.

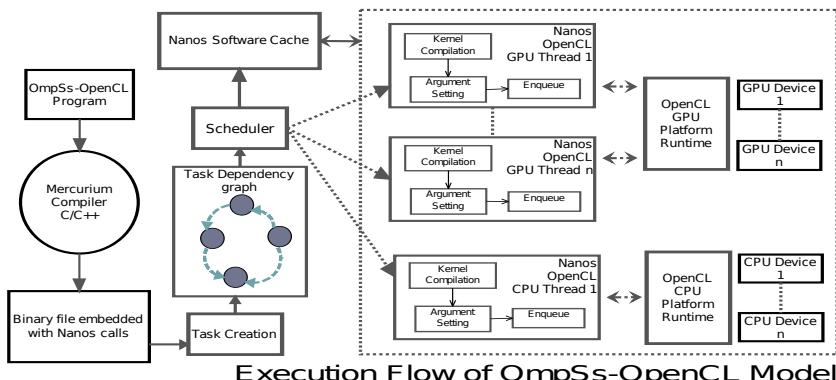


Fig. 2. Execution Flow of OmpSs-OpenCL

Nanos-OpenCL Thread Model. The Nanos-OpenCL thread is built over the OpenCL runtime. This thread is responsible for receiving the task for execution from the scheduler. The thread carries out task compilation, kernel argument setting and enqueue for execution (figure 2). During kernel enqueue, the runtime uses the *NDRange dimensions* (kernel launch parameters) from the user-defined NDRange clause of the task and launches it using *clEnqueueNDRangeKernel* [11]. In addition, each OpenCL device is associated with a Nanos-OpenCL thread (eg., 2 Nanos-OpenCL-GPU threads for 2 GPUs available). This approach is employed to provide parallel services to multiple devices under the same platform. Furthermore, if the same task/kernel code using different data is scheduled to each of the 2 devices in a platform for execution, (Eg: 2 GPUs using Nvidia’s OpenCL runtime) the runtime makes sure that the task is not compiled twice for the same platform. Moreover, each thread operates on its own OpenCL *CommandQueue*[11] specific to its device.

Nanos-OpenCL Memory Model. The Nanos Software Cache manages data transfers in and out of devices. The software cache is linked with both CPU and GPU OpenCL packages in order to carry out data transfers respectively. Once a task is scheduled to its targeted device the scheduler informs the cache engine with required information to initiate respective transfers for the task into the device. The cache system uses the task dependency information from the runtime in order to initiate transfers based on the locality of its required data.

The Nanos software cache incorporates data locality principles into its transfer management. In general, the software cache system tries to minimize the total number of transfers from memory to its devices and vice-versa. In listing 1.2, consider stream benchmark with 4 different tasks all targeted towards GPU execution exhibits different data dependence across one another. With all the tasks targeted for GPU execution, total number of transfers required for application execution is 10, 6 input and 4 output transfers according to its *#pragma* specification. If locality of each data source is considered before transfers, the number of transfers can be brought down to 4(1 input and 3 output). To elaborate, *copy_task* input, *a* vector is transferred as input and its output *c* is not transferred back as it provides the input for dependent *scale_task*. Similarly, the other 2 tasks also receive its data from its predecessor. The output transfers for *scale_task*, *add_task* and *triad_task* are carried out for *b*, *c*, *a* vectors respectively. Figure 10 shows the difference in total data transfer timing for both CPU and GPU with and without data locality for two different problem sizes of stream benchmark. With an average of more than 50% reduction in the transfer time can be understood from Figure 10. We agree it is not possible for the cache system to optimize data transfer timing of this order for all applications but we can guarantee that it cannot cause any overhead. With the runtime system orchestrating data transfers and execution of the tasks, the synchronization needed to maintain data flow and program correctness are also incorporated within it. With GPUs spending more time in data transfers due the PCI bottleneck[9], it is very crucial to hide transfer latency in order to achieve better performance. We realize this by incorporating task-prefetching technique into our runtime. In

this technique, runtime does not wait for the executing task to finish but communicates with the scheduler to prefetch the next available task. When ready tasks are available for scheduling with no data dependencies with the executing task, the Nanos-OpenCL thread picks it thereby initiating the cache system to perform the required data transfers into the device. Such transfers overlap with the execution of the current task achieving communication-computation overlap. Also, the prefetcher only fetches data if it can fit it in the available OpenCL allocation space, otherwise it does not prefetch. With Nanos-OpenCL threads working in parallel, all available devices including CPUs and GPUs can prefetch in parallel. In the next section, we investigate on scalability of OmpSs-OpenCL tasks on multiple GPUs with task-prefetching.

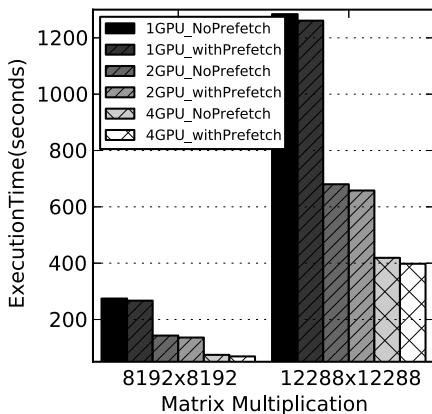


Fig. 3. Matrix Multiplication

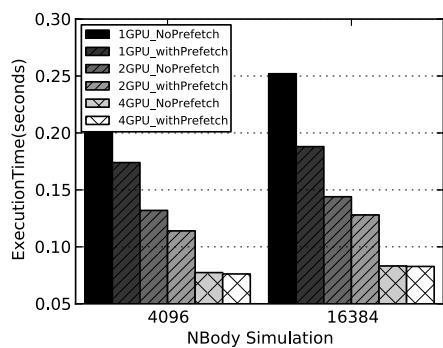


Fig. 4. NBody Simulation

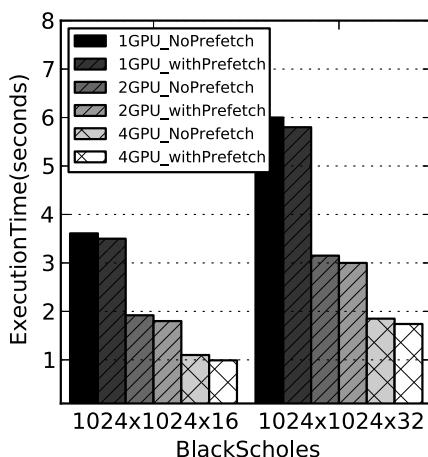


Fig. 5. BlackScholes

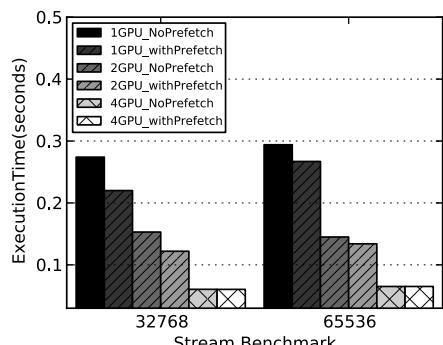


Fig. 6. Stream Benchmark

4 Scalability on Multiple GPUs

We experimented with 4 benchmarks namely, Matrix multiplication, Nbody simulation, Stream and Blackscholes with two different problem sizes for scalability on four Nvidia Tesla C2070 GPUs each having 448 CUDA cores with CUDA compiler driver V5.5. The four benchmarks are written in OmpSs-OpenCL with computations partitioned into parallel tasks in order to exploit the available 4 GPUs. 12288x12288 Matmul is partitioned into 27 tasks with each task computing 4096x4096(blocksize) and 8192x8192 into 64 tasks with each task computing on 2048x2048 block matrix size. BlackScholes is partitioned into 64 tasks and Nbody into 4 tasks. All three benchmarks are compute bound whereas Stream, a memory bandwidth limiting benchmark consist of 4 tasks in itself(add,copy, scale and triad). Original OpenCL benchmarks were obtained from [1][5]. With the runtime facilitating prefetching of tasks, our experiments also demonstrate the benefit in overlapping computation with communication. Figures 3, 4, 5 and 6 show the scalability of benchmarks on four GPUs with and without task-prefetching. All 4 benchmarks experience good scalability with an average speedup of 4x compared to its execution on a single GPU.

Using task-prefetching, matrix multiplication experiences notable gain in performance. With both problem sizes matmul gains an average of 5% in performance. To illustrate, for 12288x12288 matmul with 27 tasks, 80% of the total time taken for transferring input data is overlapped with the computation when benchmarked using 4 GPUs. The data transfers of the first 4 tasks getting scheduled to the 4 GPUs cannot be overlapped but the rest of 23 task inputs are prefetched with effective realization of computation-communication overlap for matmul. Table 1 depicts in detail percentages of overlapping experienced in both matmul and blackscholes for 4 GPUs. In Stream and Nbody benchmarks, prefetching with 4 GPUs does not provide improvements as it is partitioned into 4 tasks only. On an average, using task-prefetch there is 5% gain with matmul and BlackScholes, 15% with NBody and 10% with Stream Benchmark compared to experiments without prefetching using 1,2 and 4 GPUs repectively. With this evaluation, we show OmpSs-OpenCL support for multiple GPUs with parallel execution of tasks across them realizing good scalability.

Table 1. Task Pre-Fetch Gain

Application for 4 GPUs	Total Transfer time	Overlapped	%Overlapped
Matmul 12288x12288 (27 Tasks)	24.8 sec	19.6 sec	80%
Matmul 8192x8192 (64 tasks)	7.2 sec	6.6 sec	90%
Black Scholes 16 (64 tasks)	134 ms	125 ms	93%
Black Scholes 32 (64 tasks)	200 ms	184 ms	92%

5 Scheduling

With OmpSs-OpenCL supporting parallel execution of tasks on hybrid CPU-GPU environment there arises a chance to schedule them efficiently in order to use the resources in the best possible way. Code portability in OpenCL allows kernels to run on any OpenCL compliant platform. The reason for having the OmpSs-OpenCL task as a pure OpenCL kernel is to make it adaptive, so that it can be scheduled to any other device (other than the *targetdevice*) based on machine availability. We present two different scheduling techniques for OmpSs-OpenCL model described in the following subsections.

5.1 Static Scheduling (S)

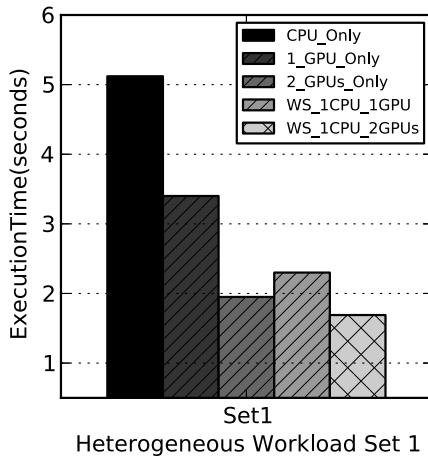
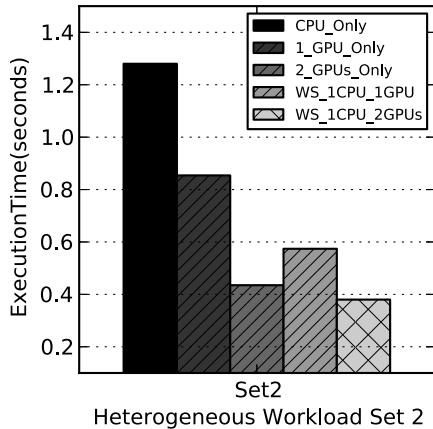
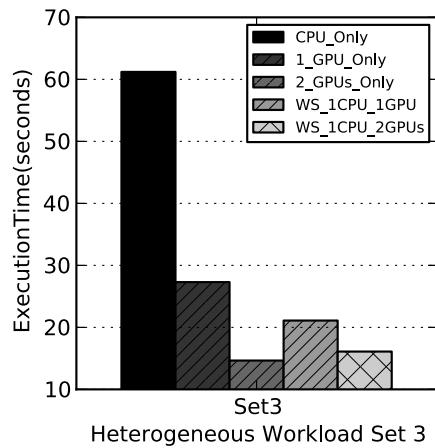
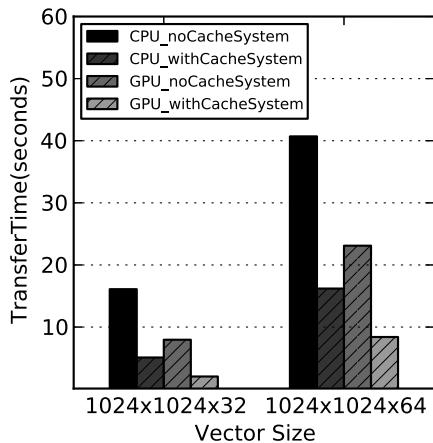
Static scheduling is a straightforward scheduling mechanism wherein the task is scheduled to the device mentioned using the *target device* clause. In case of application programmer having incorporated device-specific optimizations into the task/kernel, this approach would be implicit to go forward with. Applications which can be partitioned into tasks suited specifically to CPUs or GPUs using optimizations like vectorization, device specific *workGroupSize*, total number of *workGroups*, loop unrolling and local memory optimizations are advised to be statically scheduled. In these scenarios, static scheduling would bring out the best performance from their corresponding devices.

5.2 Work-Stealing Scheduling (WS)

The concept of work-stealing is essentially executing a task with clause *target device (clcpu)* on a GPU system and vice-versa. In this mode the scheduler assigns the task which is ready for execution to the first Nanos-OpenCL thread which goes idle. When the goal is to utilize all available devices in the environment, work-stealing scheduling would be very pertinent. However, work-stealing strategy can degrade performance, if the data needed by a task has to be transferred across devices for execution. The scheduler checks with the Nanos software cache engine for data locality before the task is scheduled. For example: given that a OpenCL-CPU is idle, if task B consumes output data from task A being executed on a GPU, the scheduler assigns task B to the GPU in order to save time from unwanted data transfer(GPU→Host→CPU). So dependent tasks are scheduled to the same device ensuring data locality, hence, preserving the application from a unwanted data transfer.

5.3 Evaluation

We evaluate both static and work-stealing techniques using MinoTauro supercomputer, a multi-GPU system running Linux with two Intel Xeon E5649 6-Core at 2.53 GHz and two NVIDIA GPUs M2090 with 512 CUDA cores. To the best of our knowledge, benchmark suites for hybrid environments composing CPUs and multiple discrete GPUs are yet to be available. Thus, for our evaluation,

**Fig. 7.** Workload Set 1**Fig. 8.** Workload Set 2**Fig. 9.** Workload Set 3**Fig. 10.** Nanos Cache System

we have developed heterogeneous workload with 8 different benchmarks written using OmpSs-OpenCL model. Original OpenCL benchmarks are obtained from [1][5].

For our experiments, we create 3 heterogeneous workloads namely, set 1,2,3 computing with different data-sizes. All benchmarks in the workload are independent from one another, providing possible parallel execution of tasks allowing us to investigate static and work-stealing techniques. In particular, Blocked matmul(8 independent tasks), Bitonic sort(independent tasks based on problem size) and Stream(4 dependent tasks) are partitioned, whereas other benchmarks

including Blackscholes, Reduction, Nbody simulation, Convolution and Transpose compute using only a single task. Figure 7, 8 and 9 shows the evaluation of the three workload sets on our heterogeneous platform, with task-prefetch facility activated. Table 2 gives the overview of the workload characterization with its problem size, execution timings on both devices and scheduling decisions taken using work-stealing technique.

Table 2. Heterogeneous Workload Characterization

Benchmark	Set	Problem Size	CPU Execution Time(ms)	GPU Execution Time(ms)	Sch Decision (WS-1Gpu)	Sch Decision (WS-2Gpus)
Blocked Matmul	1	1024x1024	259.75	109.1	Cpu and Gpu	Cpu and Gpu
	2	512x512	27.4	9.1	Cpu and Gpu	Cpu and Gpu
	3	2048x2048	8123.3	3367.6	Cpu and Gpu	Cpu and Gpu
Matrix Transpose	1	2048x2048	5.2	1.0	Gpu	Gpu
	2	1024x1024	2.1	0.257	Gpu	Gpu
	3	512x512	1.24	0.0714	Cpu	Gpu
Black Scholes	1	1024	0.0622	0.009	Gpu	Gpu
	2	4096	0.138	0.11	Gpu	Gpu
	3	16384	0.215	0.0170	Gpu	Gpu
Bitonic Sort	1	4096	0.089	0.008	Cpu and Gpu	Cpu and Gpu
	2	512	0.0377	0.016	Cpu and Gpu	Cpu and Gpu
	3	1024	0.034	0.012	Cpu and Gpu	Cpu and Gpu
Convolution	1	256	0.582	0.0071	Gpu	Gpu
	2	1024	0.88	0.040	Cpu	Gpu
	3	4096	1.4	0.0544	Gpu	Gpu
NBody	1	4096	22.8	6.02	Gpu	Gpu
	2	512	0.569	0.715	Gpu	Gpu
	3	1024	2.89	1.42	Gpu	Gpu
Stream	1	1024	0.12	0.028	Gpu	Gpu
	2	4096	0.18	0.024	Gpu	Gpu
	3	8192	0.310	0.0412	Gpu	Gpu
Reduction	1	4096	0.571	0.010	Cpu	Gpu
	2	1024	0.488	0.010	Gpu	Gpu
	3	16384	1.01	0.0121	Cpu	Gpu

Set 1 with benchmarks using both larger and smaller problem sizes, work-stealing with both 1 and 2 GPUs provides performance gain for the workload. In our evaluation (Figure 7, 8 and 9), the performance gain calculated is in comparison with the execution time of workloads using static schedule with 1 and 2 GPUs. Using WS¹ 1CPU-1GPU experiences almost 30% performance gain with Reduction benchmark running on the CPU and both Blocked matmul and

¹ WS-Work-Stealing Scheduling Technique.

Bitonic sort with multiple tasks gets shared across both devices. In WS 1CPU-2GPUs the gain is 12% with Blocked matmul and Bitonic sort executed among both devices. Set 2 workload consist of benchmarks with small sizes provides gain of 33% and 10% for 1CPU-1GPU and 1CPU-2GPU repectively. Convolution gets scheduled to CPU with WS 1CPU-1GPU and both bitonic and blocked matmul gets scheduled to both devices in both work-stealing cases.

Set 3 workload consist of benchmarks working with larger data-sets. WS with 1CPU-1GPU provides 22% performance gain with Reduction and transpose undergoing CPU execution. However, WS with 1CPU-2GPUs experiences performance loss of 13%(Figure 9). This is to say static schedule of Set 3 to 2 GPUs perform better compared to WS with 1CPU and 2 GPUs. In this case, adoption of WS technique is not favorable on the performance front although it utilizes all available devices. Apparently, tasks inherently suited for GPU can be scheduled to CPU. In this case, Blocked matmul exhibits significant difference in execution time with CPU execution being much slower compared to GPU and is scheduled to both GPU and CPU accounts for this loss in performance. Moreover, with 2 powerful GPUs in WS mode enhances tasks to be scheduled to them compared to a single available CPU thereby making device count and its characteristics a crucial parameter to be considered during scheduling.

From Table 2, we can learn that all benchmarks for different problem sizes work faster in the GPUs. Currently, considering the standard that GPUs are not standalone systems and CPUs are inherent in heterogeneous systems, utilizing CPUs effectively becomes crucial. With GPUs and CPUs becoming more powerful and power efficient and with every generation, it is imperative for the user to use both devices in a hybrid environment as HPC components. Moreover, with supercomputing design going heterogeneous, the design of algorithms and workloads involving mixture of varied computations exhibiting different types of parallelism would be critical. This puts forth a need to have a single programming model to harness these resources. OmpSs-OpenCL model provides this facility with complete abstraction to the user programmer. Further, in order to effectively use the resources we plan to extend our work focusing on development of an optimal scheduling methodology. With extensive analysis of our workloads, we would like to characterize parameters like computational complexity of tasks, data transfer timings, runtime status, device load/contention [10] which comprehensively define the best possible device for scheduling based on the execution environment.

6 Related Work

With GPUs becoming prominent in HPC domain, lots of research groups have focussed on their programmability. [13], SnuCL provides OpenCL framework extending original OpenCL semantics for heterogeneous cluster environment. This framework does not offer an abstraction to the users to make OpenCL programming easier unlike OmpSs-OpenCL. In [2], StarPU run-time environment offers programming language extensions supporting task-based model. StarPU

model expects user to know OpenCL API calls (eg: Kernel launch and argument setting) in order to write applications using them. Whereas, OmpSs-OpenCL provides complete abstraction to OpenCL API for programming heterogeneous systems. CAPS HMPP [6] is a toolkit with a set of compiler directives, tools and runtime supporting parallel programming in C and Fortran. It is based on hand-written codelets defining the parallel regions to be run on accelerators. OpenACC [12] high level programming API describes a collection of compiler directives used to specify regions of code for offloading from a host CPU to an attached accelerator but compared to OpenCL it is not yet adopted among all microprocessor and accelerator vendors. In [17], the author describe a wrapper model *clUtil* which simplifies OpenCL API. It provides a wrapper to OpenCL, where the user can skip OpenCL constructs by replacing appropriate *clUtil* calls in order to do its functionalities. Whereas, OmpSs-OpenCL model offers users to write simple sequential style programming with no complex calls to the runtime. In general, OmpSs-OpenCL programming model offers a high-level uniform programming approach. With current supercomputers already following hybrid designs, we believe that OmpSs-OpenCL model can be very effective in realizing productive supercomputing.

7 Conclusion and Future Work

In this paper, we upgrade OmpSs-OpenCL programming model to support parallel execution of tasks across CPU-GPU hybrid systems. We discuss the implementation of Nanos runtime system which plays a key role in realizing heterogeneous computing. Along with this support, we present static and work-stealing scheduling techniques. Static scheduling is user-assisted scheduling, whereas work-stealing utilizes all the available devices in the system, in turn increasing its throughput. We evaluated scalability of 4 benchmarks using OmpSs-OpenCL model on 4 GPUs providing an average speedup of 4x compared to one GPU. We used three sets of heterogeneous workloads to investigate parallel execution of tasks on hybrid platform using both static and work-stealing strategies. With OmpSs-OpenCL model being extensible, we look forward to support heterogeneous devices like Intel Xeon-Phi and FPGAs. Moreover, we plan to use the results from our evaluation to devise an optimal dynamic scheduling algorithm for CPU-GPU hybrid systems.

Acknowledgement. We thankfully acknowledge the support of the European Commission through the TERAFLUX project (FP7-249013) and the HiPEAC-2 Network of Excellence (FP7/ICT 217068), the support of the Spanish Ministry of Education (TIN-2007-60625, TIN-2012-34557, CSD2007-00050 and FI program) and the Generalitat de Catalunya (2009-SGR-980).

References

1. AMD. Amd sdk examples, <http://developer.amd.com/tools-and-sdks/heterogeneous-computing/>
2. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.-A.: Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience* 23(2), 187–198 (2011)
3. Ayguadé, E., Badia, R.M., Igual, F.D., Labarta, J., Mayo, R., Quintana-Ortí, E.S.: An extension of the starss programming model for platforms with multiple gpus. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 851–862. Springer, Heidelberg (2009)
4. Barcelona Supercomputing Center. The nanos group site: The mercurium compiler, <http://nanos.ac.upc.edu/mcxx>
5. Danalis, A., Marin, G., McCurdy, C., Meredith, J.S., Roth, P.C., Spafford, K., Tipparaju, V., Vetter, J.S.: The scalable heterogeneous computing (shoc) benchmark suite. In: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, pp. 63–74. ACM (2010)
6. Dolbeau, R., Bihan, S., Bodin, F.: Hmpp: A hybrid multi-core parallel programming environment. In: Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007) (2007)
7. Duran, A., Ayguadé, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X., Planas, J.: Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters* 21(2), 173–193 (2011)
8. Elangovan, V.K., Badia, R.M., Parra, E.A.: OmpSs-openCL programming model for heterogeneous systems. In: Kasahara, H., Kimura, K. (eds.) LCPC 2012. LNCS, vol. 7760, pp. 96–111. Springer, Heidelberg (2013)
9. Gregg, C., Hazelwood, K.: Where is the data? why you cannot debate cpu vs. gpu performance without the answer. In: 2011 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 134–144. IEEE (2011)
10. Grewe, D., Wang, Z., O’Boyle, M.F.P.: Opencl task partitioning in the presence of gpu contention
11. Khronos OpenCL Working Group et al.: The opencl specification. In: Munshi, A. (ed.) (2008)
12. OpenACC Working Group et al. The openacc application programming interface (2011)
13. Kim, J., Seo, S., Lee, J., Nah, J., Jo, G., Lee, J.: Snocl: an opencl framework for heterogeneous cpu/gpu clusters. In: Proceedings of the 26th ACM International Conference on Supercomputing, pp. 341–352. ACM (2012)
14. McCalpin, J.D.: A survey of memory bandwidth and machine balance in current high performance computers. *IEEE TCCA Newsletter*, 19–25 (1995)
15. Munshi, A., Gaster, B., Mattson, T.G., Ginsburg, D.: OpenCL programming guide. Pearson Education (2011)
16. CUDA Nvidia. Programming guide (2008)
17. Webber, R.: Clutil-making opencl as easy to use as cuda (website)

Automatic Exploration of Potential Parallelism in Sequential Applications

Vladimir Subotic¹, Eduard Ayguadé^{1,2}, Jesus Labarta^{1,2}, and Mateo Valero^{1,2}

¹ Barcelona Supercomputing Center, Barcelona, Spain

`vladimir.subotic@bsc.es`

² Universitat Politècnica de Catalunya, Barcelona, Spain

Abstract. The multicore era has increased the need for highly parallel software. Since automatic parallelization turned out ineffective for many production codes, the community hopes for the development of tools that may assist parallelization, providing hints to drive the parallelization process. In our previous work, we had designed Tareador, a tool based on dynamic instrumentation that identifies potential task-based parallelism inherent in applications. Also, we showed how a programmer can use Tareador to explore the potential of different parallelization strategies. In this paper, we build up on our previous work by automating the process of exploring parallelism. We have designed an environment that, given a sequential code and configuration of the target parallel architecture, iteratively runs Tareador to find an efficient parallelization strategy. We propose an autonomous algorithm based on simple metrics and a cost function. The algorithm finds an efficient parallelization strategy and provides the programmer with sufficient information to turn that parallelization strategy into an actual parallel program.

Keywords: automatic parallelization, potential parallelism, OmpSs.

1 Introduction

For decades, microprocessors have been improving their performance following Moore’s law without requiring major changes in the applications. In essence, the performance improvements relied on both architectural techniques that improve ILP (instruction-level parallelism) and compilers that optimize the code for each target architecture. Unfortunately, the improvements achieved by ILP had entered stagnation.

On the other hand, multicore processor architectures are now the norm in high-performance processors. Multicore processors have introduced the need to re-design applications in order to utilize the increasing number of available cores. Applications that before were running sequentially, now must be parallelized in order to efficiently harness the full potential of multicore processors. However, parallelizing existing applications is not an easy task. As the software community struggles to fulfill this demand, the gap between parallel hardware and sequential software keeps growing.

It is believed that neither the compiler nor the hardware itself will automatically detect and exploit the parallelism needed to feed current and future multi-/many-core architectures. Despite decades of research efforts [4,1,16] on auto-parallelization, and the inclusion of auto-parallelization features in some commercial compilers [2], the experience have shown that they have very limited applicability. In the current scenario, in which systems (from mobile to desktop/laptop and servers) are based mostly on parallel architectures, programmers must use explicit parallel programming techniques.

To help in the process of parallelization, the community has developed several tools to assist the parallelization process (see Section 7). These tools usually target a specific parallel programming model or language and/or impose constraints of the possible strategies to explore (e.g. loops).

In our prior work, we proposed Tareador [15] as a tool to analyze the potential parallelism inherent in applications. We also described an iterative top-down trial-and-error process to find suitable parallelization strategies. However, the presented process relied strongly on programmer’s experience to guide the search.

In this paper, we propose an automatic exploration of parallelization strategies. Our goal is to formalize the programmers experience into an autonomous algorithm that can find an effective task decomposition of a sequential code. More specifically, our work provides the following contributions:

1. **Definition of a set of metrics and heuristics** that drive the automatic exploration of parallelization strategies. Our heuristics mandate the policy of refining decomposition in order to increase parallelism, as well as the end of the iterative exploration. The proposed metrics are parameterizable so they can be customized according to the targeted sequential code.
2. **Design of an environment** that leverages Tareador to automatically explore parallelism in sequential codes. The designed tool-chain iteratively tests various task decompositions, illustrating a reasonable exploration path for exposing parallelism inherent in the code. Furthermore, the environment offers visualization of the parallel execution of the tested decompositions.

The paper is organized as follows. Section 2 briefly summarizes the Tareador environment. Section 3 describes the proposed algorithm (metrics and heuristics) that autonomously explores task decomposition strategies and Section 4 presents the implemented environment. In Section 5 we present the results obtained for a set of simple applications, while in Section 6 we discuss how our approach could be applied to realistic workloads. Finally, Section 7 describes some related work and environments and Section 8 concludes the paper.

2 Background: Tareador

Tareador [15] is a tool for assisted parallelization of sequential applications. Using the Tareador API, the programmer annotates the sequential code to propose a task decomposition. Then, the tool (implemented as a Valgrind [12] plugin) dynamically instruments the code in order to collect all memory accesses within

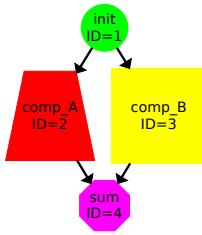
```
tareador_start_task("init");
A = random_buf ();
B = reset_buf ();
tareador_end_task();

tareador_start_task("comp_A");
compute(A);
tareador_end_task();

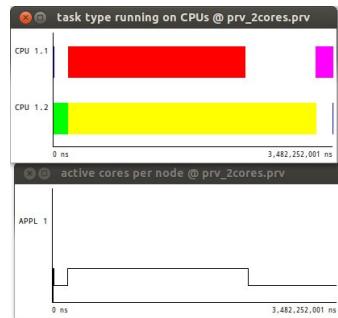
tareador_start_task("comp_B");
compute(B);
tareador_end_task();

tareador_start_task("sum");
res += sum (A);
res += sum (B);
tareador_end_task();
```

(a) Input code



(b) task graph



(c) Paraver views

Fig. 1. Tareador instrumentation

each specified task. Based on the collected accesses, Tareador derives the data dependencies among the tasks and estimates the potential parallelism of the task decomposition. In our prior work [15] we demonstrated how a programmer can use Tareador to iteratively explore the task decomposition space and find a decomposition that exposes sufficient parallelism to efficiently deploy multi-core processors. Depending on the application (granularity of tasks, number of dependencies, ...), Valgrind instrumentation introduces the slowdown of $200x$ - $1000x$ compared to the native sequential execution of the target application.

Tareador provides to a programmer a simple and flexible API to propose how a sequential code could be decomposed into tasks. Namely, the programmer invokes *tareador_start_task* to mark the beginning of a task, and *tareador_end_task* to mark the end of a task. The interface allows specification of any arbitrary task decomposition, even if the targeted code is badly structured or recursive (nesting of tasks is supported). No other refactoring of the targeted sequential code is needed. Figure 1a illustrates a simple code with Tareador annotations.

As an evaluation of the proposed decomposition, Tareador provides to the programmer two outputs: the dependency graph of all tasks; and the simulation of the potential parallel execution. Figure 1b shows the task graph for the previous example: a node represents a dynamic task instance and an edge represents a dependency between two task instances. In this example, the graph suggests to the programmer that there is potential concurrency only between task *comp_A* and task *comp_B*. Moreover, Figure 1c shows the timeline for the simulated parallel execution on a target processor with 2 cores. The upper timeline (horizontal axis is time) shows which task executes on each core and the lower timeline shows the number of active cores throughout the simulated execution. The same colors are used to represent matching tasks in the graph and in the timeline, helping the programmer to identify potential bottlenecks in the current task decomposition (for example, in this case to observe the load imbalance that occurs in the parallel execution of tasks *comp_A* and *comp_B*).

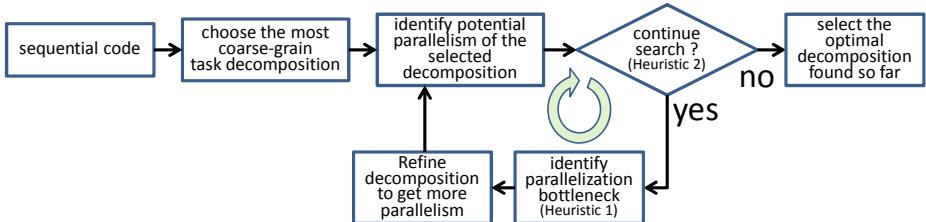


Fig. 2. Algorithm for exploring parallelization strategies

3 Automatic Exploration of Parallelization Strategies

The automatic exploration of parallelization strategies is based on: evaluating parallelism of various decompositions; collecting key parameters that identify the parallelization bottlenecks; and refining decompositions in order to increase parallelism. The search algorithm is illustrated in Figure 2. The inputs of this algorithm are the original unmodified sequential code and the number of cores in the target platform. The search algorithm passes through the following steps:

1. Start from the most coarse-grain task decomposition, i.e. the one that considers the whole main function as a single task.
2. Perform an estimation of the potential parallelism of the current task decomposition (the speedup with respect to the sequential execution).
3. If the exit condition is met (*Heuristic 2*), finish the search.
4. Else, identify the parallelization bottleneck (*Heuristic 1*), i.e. the task that should be decomposed into finer-grain tasks.
5. Refine the current task decomposition in order to avoid the identified bottleneck. Go to step 2.

In the following sections, we further describe the design choices made in designing the mentioned heuristics. Nevertheless, first we must define a more precise terminology. Primarily, we must make a clear distinction between a **task type** (function that is encapsulated into task) and a **task instance** (dynamic instance of that function). For instance, if function *compute* is encapsulated into a task, we will say that *compute* is a **task type**, or just a **task**. Conversely, each

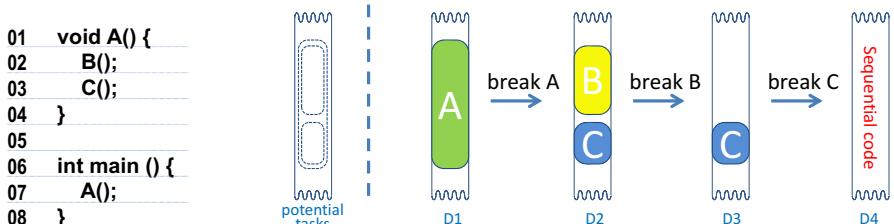


Fig. 3. Iterative refinement of decompositions

instantiation of *compute* we will call a **task instance**, or just an **instance**. A task instance is atomic and sequential, but various instances (of same or different task type) can execute concurrently among themselves.

Also, we will often use a term **breaking a task** to refer to the process of transforming one task into more fine-grain tasks. For example, Figure 3 illustrates decomposition refining in a case of a simple code. The process starts with the most coarse-grain decomposition (*D1*) in which function *A* is the only task. By breaking task *A*, we obtain decomposition *D2* in which *A* is not a task and instead its direct children (*B* and *C*) become tasks. If in the next step we break task *B*, assuming that *B* contains no children tasks, *B* will be serialized (i.e. *B* is not a task anymore and its computation becomes a part of the sequential execution). Similarly, the next refinement serializes task *C* and leads to the starting sequential code. At this point, no further refinement is possible, so the iterative process naturally stops.

3.1 Heuristic 1: Which Task to Break

In the manual search for an efficient decomposition, the programmer decides which task is the parallelization bottleneck. The practice shows that the bottleneck task is often one of the following:

1. **the task whose instances have long duration**, because a long instance may cause significant load imbalance.
2. **the task whose instances have many dependencies**, because an instance with many dependencies may be a strong synchronization point.
3. **the task whose instances have low concurrency**, because an instance with low concurrency may prevent other instances to execute in parallel.

Our goal is to formalize this programmer experience into a simple set of metrics that can lead an autonomous algorithm for exploring potential task decompositions. The goal is to define a **cost function** for task type *i* as:

$$\overline{t}_i = \overline{l_i(p_l)} + \overline{d_i(p_d)} + \overline{c_i(p_c)} \quad (1)$$

where $\overline{l_i}$, $\overline{d_i}$ and $\overline{c_i}$ are functions that calculate the partial costs related to tasks' length, dependencies count and concurrency level. On the other hand, parameters p_l , p_d and p_c are empirically identified parameters that tune the weight of each partial cost within the overall cost. The rest of this section further describes the operands from Equation 1.

Metric 1: Task Length Cost. A task type that has long instances is a potential parallelization bottleneck. Thus, based on the length of instances, we define a metric called length cost of a task type. Length cost of some task type is proportional to the length of the longest instance of that task. Therefore, if task *i* has instances whose lengths are in the array T_i , the length cost of task *i* is:

$$l_i = \max(t), t \in T_i \quad (2)$$

Furthermore, we define a normalized length cost of task i as:

$$\overline{l_i(p)} = \frac{(l_i)^p}{\sum_{j=1}^N (l_j)^p}, \quad 0 \leq p \leq \infty, \quad 1 \leq i, j \leq N \quad (3)$$

where the control parameter p is used to tune the distribution of normalized costs (explained later in this section).

Metric 2: Task Dependency Cost. A task type that causes many dependencies is another potential parallelization bottleneck. Thus, based on the number of dependencies (sum of incoming and outgoing dependencies), we define a metric called dependency cost of a task type. Dependency cost of some task is proportional to the maximal number of dependencies caused by some instance of that task. Therefore, if task i has instances whose numbers of dependencies are in the array D_i , the dependencies cost of task i is:

$$d_i = \max(z), z \in D_i \quad (4)$$

Furthermore, using a control parameter p , we define the normalized dependency cost of task i as:

$$\overline{d_i(p)} = \frac{(d_i)^p}{\sum_{j=1}^N (d_j)^p}, \quad 0 \leq p \leq \infty, \quad 1 \leq i, j \leq N \quad (5)$$

Metric 3: Task Concurrency Cost. A task type that has low concurrency is another potential parallelization bottleneck. Concurrency of some instance is determined by the overall utilization of the machine during the execution of that instance. Thus, we define concurrency cost of some task to be inversely proportional to the average number of cores that are efficiently utilized during the execution of that task. Therefore, if task i has task instances which run for time $T_{i,j}$ while there are j cores efficiently utilized, the concurrency cost of task i is:

$$c_i = \frac{\sum_{j=1}^{cores} \frac{T_{i,j}}{j}}{\sum_{j=1}^{cores} T_{i,j}} \quad (6)$$

Again, using a control parameter p , we define the normalized concurrency cost of task i as:

$$\overline{c_i(p)} = \frac{(c_i)^p}{\sum_{j=1}^N (c_j)^p}, \quad 0 \leq p \leq \infty, \quad 1 \leq i, j \leq N \quad (7)$$

Control Parameter p . Introduction of the parameter p provides the mechanism for controlling the mutual distance of the normalized costs for different tasks. For instance, let us assume that the application consists of two task instances, A and B , where A is two times longer than B . If the control parameter p_l is equal to 1, the normalized length costs for tasks A and B are 0.67 and 0.33, respectively. However, if the control parameter p_l is equal to 2, the costs for tasks A and B become 0.8 and 0.2, respectively.

Therefore, by changing parameter p of some metric, we can control the impact of that metric on the overall cost. For example, if the control parameter for length cost is 0, all task types will have the same normalized length cost, independent of the length of task instances. Thus, the length of tasks would have no impact on the overall cost. On the other hand, if the control parameter for length cost is infinite, the task type with the longest instance will have the normalized length cost of 1, while all other task types will have the normalized length cost of 0. This way, the impact of the task length on the overall cost would be maximized.

3.2 Heuristic 2: When to Stop Refining the Decomposition

The algorithm also needs a condition to stop the iterative search. Iterative search leads to fine grain decompositions that instantiate a very high number of tasks. An excessive number of tasks causes a very complex and computation intensive evaluation of the potential parallelism. Thus, to make the complete automatic search viable, we must adopt the exit condition that will prevent processing unnecessary decompositions.

To construct the Heuristic 2, we must create a system for rating the quality of a decomposition. Our basic rating system consists of two rules. First, out of all tested decompositions, the optimal decomposition is the one that achieves the highest parallelism. Second, if the optimal decomposition achieves the parallelism of s_{opt} and instantiates t_{opt} tasks, and some other decomposition i achieves the parallelism of s_i and instantiates t_i tasks, the relative quality of decomposition i compared to the optimal decomposition is:

$$\text{Quality}_i = \left(\frac{s_i}{s_{opt}} \right) \cdot \left(\frac{t_{opt}}{t_i} \right)^{\text{exp_tasks}}, \quad 0 \leq \text{exp_tasks} \leq 1 \quad (8)$$

Thus, the relative quality of some decomposition drops as the achieved parallelism drops and as the number of instantiated tasks increases. Furthermore, the parameter exp_tasks serves to tune the impact of the number of instantiated tasks.

Finally, Heuristic 2 mandates that the iterative search stops if the current decomposition has relative quality lower than some threshold value:

$$\text{Quality}_i < (Q_{threshold})^{\frac{\text{cores}}{s_{opt}}}, \quad 0 \leq Q_{threshold} \leq 1 \quad (9)$$

The right side of this expression increases with the increase of the parallelism of the optimal task decomposition. Thus, if the optimal parallelism is close to

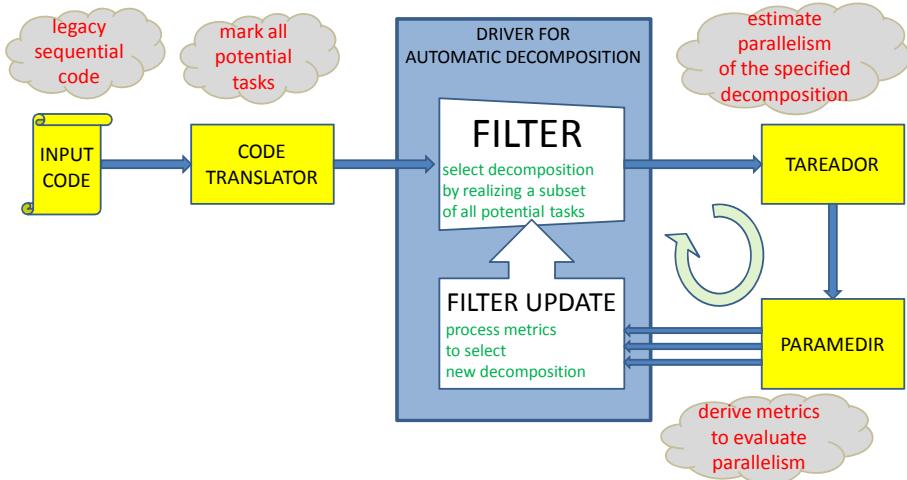


Fig. 4. Environment to automatically explore possible task decompositions

the theoretical maximum (number of cores in the target machine), finding a better decomposition is unlikely, so the algorithm should not tolerate high quality degradations. On the other hand, if the optimal found parallelism is far from the theoretical maximum, the algorithm should be more aggressive in finding a better decomposition, and therefore allow high degradations of quality.

4 Designed Environment

Our environment for automatic exploration of task decompositions (Figure 4) consists of: Tareador, Paramedir and the Driver. In addition, a source-to-source translator automatically annotates all potential tasks in the code (main function, each function call and each loop). Tareador [15] evaluates the potential parallel execution of a task decomposition. Paramedir [9], the non-graphical user interface to the Paraver [14] analysis tool, extracts performance metrics of the simulated parallel execution. Finally, the Driver is a glue that integrates all the mentioned tools in a common environment. Its important to stress that the most computation intensive processing (Valgrind instrumentation) is performed only once, and then the generated logs are used offline to test various task decompositions. The following paragraph describes this integration in more detail.

The main functionality of the Driver is to guide the iterative decomposition refinement. In each iteration, Driver specifies a *list of tasks* that compose the current task decomposition. Initially the list contains only the main function of the program. The Driver automates the process of exploring potential decompositions by guiding the environment through the following steps:

1. **Generate execution logs:** use Valgrind to dynamically instrument the application and derive memory usage logs.

2. **Select the starting decomposition:** put the whole main into one task.
3. **Estimate the parallelism of the current decomposition:** run Tareador to generate traces that describe parallelism of the current decomposition.
4. **If the exit condition is fulfilled, finish:** if the *Quality* of the current decomposition is unsatisfactory (Heuristic 2), end the search.
5. **Else, identify the parallelization bottleneck:** process the traces with Paramedir to derive metrics that identify the bottleneck task (Heuristic 1).
6. **Refine the current decomposition to increase parallelism:** break the bottleneck task into its children tasks, if any. Update the *list of tasks* that should be included in the next decomposition.
7. **Proceed to the next iteration:** go to step 3.

5 Experiments

Our experiments explore possible parallelization strategies for four well-known applications (Jacobi, HM transpose, Cholesky and LU factorization). We select a homogeneous multi-core processor as the simulated target platform. The goal of our experiments is to show that the proposed search algorithm, metrics and heuristics can find decompositions that provide sufficient parallelism.

Table 1. Empirically identified parameters of the automatic search

p_l	p_d	p_c	exp_{tasks}	$Q_{threshold}$
1	1	3	$\log_{10}1.5$	0.75

Table 1 lists the empirically identified values for the parameters defined in Section 3. As already mentioned, the total cost function is a sum of length, dependency and concurrency cost (Equation 1). Moreover, since our initial experiments showed that concurrency criterion prevails very rarely, we decided to increase the weight of the concurrency cost. Furthermore, in Equation 8, we set the parameter exp_{tasks} so that the increase of task instances by a factor of 10 is equivalent to the decrease of speedup by a factor of 1.5. Finally, in Equation 9, parameter $Q_{threshold}$ was set empirically to allow sufficient quality degradation for a flexible search.

5.1 Illustration of the Iterative Search

This subsection illustrates our algorithm on a couple of small examples. We start from the example presented in Section 2. Table 2 enlists the tasks costs for example from Figure 1. Since tasks *comp_B* is the longest (Figure 1c), it gets the highest length cost. On the other hand, because of their very short length, tasks *init* and *sum* get very low length costs. The Table also shows that all tasks have the same dependency cost. This is because, for each task, the sum of incoming

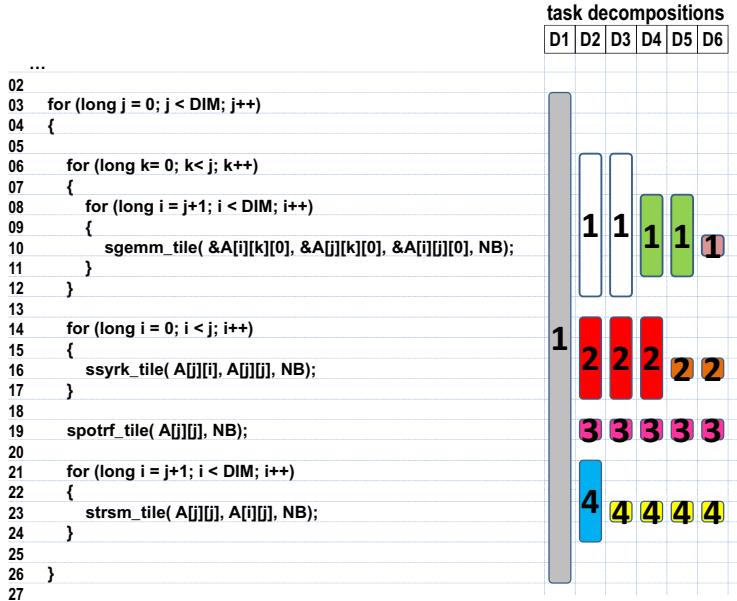


Fig. 5. Cholesky: decomposition of the code into tasks

and outgoing dependencies is equal to 2 (Figure 1b). Figure 1c also explains the tasks' concurrency costs. Since during the whole execution of task *comp_A* both cores are utilized, this task has the lowest concurrency cost. On the other hand, during the execution of tasks *init* and *sum* there is only one core active, so these tasks have the highest concurrency cost. Finally, task *comp_B* obtains the highest total cost of 0.90. Thus, in this example, our algorithm would identify *comp_B* as the bottleneck task, dominantly following the length criterion.

The second illustration of the algorithm uses the example of parallelizing Cholesky sequential code on a simulated machine with 4 cores. Figure 5 presents (on the left) the code of Cholesky and illustrates (one the right) how the code can be encapsulated into tasks for various decompositions (*D1-D6*). Note that marked task types (boxes with numbers) may generate multiple task instances, and that the code outside of marked tasks belongs to the *master task* (sequential part of execution that spawns worker tasks). Table 3 shows the speedup achieved in each decomposition and the costs that guide the iterative search. The algorithm starts from the most coarse-grain decomposition *D1* that puts

Table 2. Tasks costs for the example from Figure 1

init			comp_A				comp_B				sum				
$\bar{l}_i(1)$	$\bar{d}_i(1)$	$\bar{c}_i(3)$	\bar{l}_i												
0.03	0.25	0.42	0.70	0.39	0.25	0.05	0.69	0.54	0.25	0.11	0.90	0.04	0.25	0.42	0.71

Table 3. Cholesky: task costs (Heuristic 1)

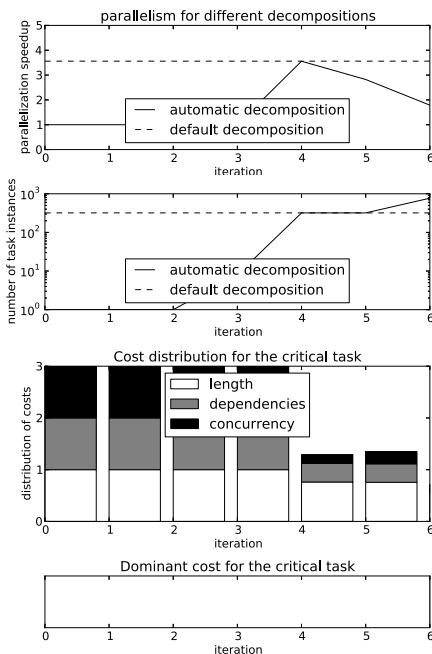
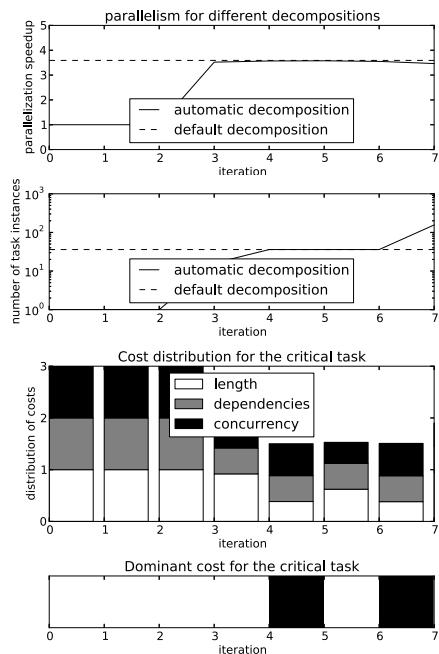
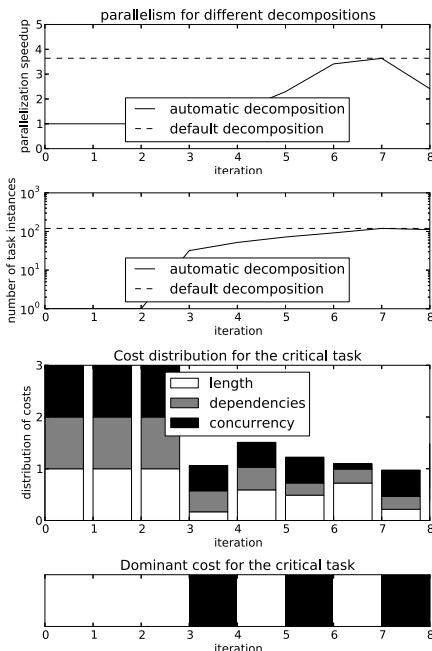
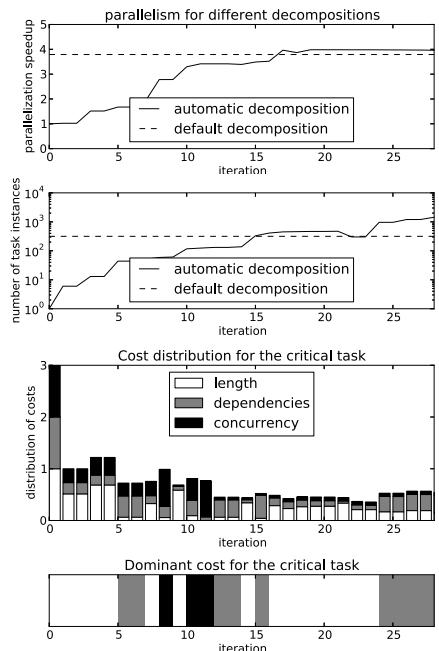
decomposition	speedup	task #1				task #2				task #3				task #4				
		$\bar{I}_i(1)$	$\bar{J}_i(1)$	$\bar{C}_i(3)$	\bar{I}_i													
D1	1.00	1.00	1.00	1.00	3.00													
D2	1.30	0.51	0.21	0.24	0.96	0.29	0.25	0.12	0.67	0.03	0.13	0.15	0.31	0.17	0.41	0.49	1.06	
D3	1.49	0.59	0.44	0.48	1.51	0.34	0.18	0.22	0.74	0.04	0.18	0.27	0.48	0.03	0.21	0.03	0.27	
D4	2.30	0.42	0.25	0.04	0.71	0.49	0.24	0.50	1.22	0.05	0.24	0.42	0.70	0.04	0.28	0.04	0.36	
D5	3.41	0.72	0.27	0.11	1.10	0.12	0.17	0.13	0.43	0.09	0.25	0.61	0.95	0.07	0.30	0.15	0.52	
D6	3.64	0.31	0.21	0.13	0.65	0.30	0.17	0.22	0.70	0.22	0.25	0.50	0.97	0.17	0.36	0.14	0.68	

the whole execution into one task. There is only one task (#1, lines 3-26), which is automatically the critical task that needs to be broken. Refining $D1$ generates decomposition $D2$ that achieves the speedup of 1.30 (Table 3) and consists of 4 different task types (Figure 5): #1 that covers the first loop; #2 that covers the second loop; #3 that covers function *spotrf_tile*; and #4 that covers the third loop. Heuristic 1 identifies task #4 (lines 21-24) as the most critical, mostly due to its high concurrency cost. Thus, the following decomposition ($D3$) breaks the task #4 and obtains the parallelism of 1.49. In $D3$, the algorithm identifies task #1 (lines 6-12) as the bottleneck (due to its high length). Further iterations of the algorithm pass through decompositions $D4$, $D5$ and $D6$ that provide speedups of 2.30, 3.41 and 3.64, respectively.

5.2 Results

This subsection presents the results obtained by applying our algorithm on a set of applications. For each application, we present four plots that illustrate the process of automatic task decomposition. The first plot presents the parallelism of all tested decompositions – the speedup over the sequential execution of the application. The second plot shows the number of task instances generated by each decomposition. Also, the first two plots show the parallelism and the number of instances in the *reference task decomposition* (the decomposition selected and implemented by an expert OmpSs[6] programmer). The third plot presents the cost distribution for the bottleneck task of each iteration. Finally, the fourth plot shows the most dominant cost for the bottleneck task.

The proposed search algorithm finds decompositions with very high parallelism, often finding the decomposition manually selected by an expert programmer. The algorithm finds the reference decomposition for Jacobi and HM transpose (Figures 6 and 7) in iterations 4 and 5, respectively. In these two applications, the algorithm bases its decisions mainly on the length criterion. The algorithm also finds the reference decomposition for Cholesky in iteration 7 (Figure 8). However, in order to get to this decomposition, the algorithm refines decompositions based on the concurrency criterion in iterations 3 and 5. In all three applications, soon after finding the reference decomposition, the algorithm passes through decompositions that activate the mechanism for stopping the search (Heuristic 2).

**Fig. 6.** Jacobi on 4 cores**Fig. 7.** HM transpose on 4 cores**Fig. 8.** Cholesky on 4 cores**Fig. 9.** Sparse LU on 4 cores

Sparse LU (Figure 9), as the most complex of the studied applications, demonstrates the power of our search. Compared to the previous codes, Sparse LU forces the algorithm to use various bottleneck criteria through the exploration of decompositions. It is interesting to note that the search finds a wide range of decompositions (iterations 17-28) that provide higher parallelism than the reference decomposition. In this case, it is unclear which of these decompositions is the optimal one. Quantitative reasoning suggests that the optimal task decomposition is the one that provides highest parallelism with the lowest number of created task instances. Following this reasoning, the optimal decomposition (iteration 22) achieves the speedup of 3.98 with the cost of 301 instantiated tasks (note the sudden drop in the number of task instances). On the other hand, qualitative reasoning suggests that, within a set of decompositions that provide a similar parallelism generating a similar number of instances, the optimal decomposition is the one that is the easiest to express using semantics offered by the target parallel programming model. For example, our algorithm may find a decomposition that extracts very irregular parallelism that cannot be expressed using a fork-join programming model. In that case, it is programmers responsibility to, out of few offered efficient task decompositions, identify the one that can be straightforwardly implemented using a specific programming model.

It is also interesting to study how the algorithm adapts to the target parallel machine. Changing the parallelism of the target machine changes the simulation

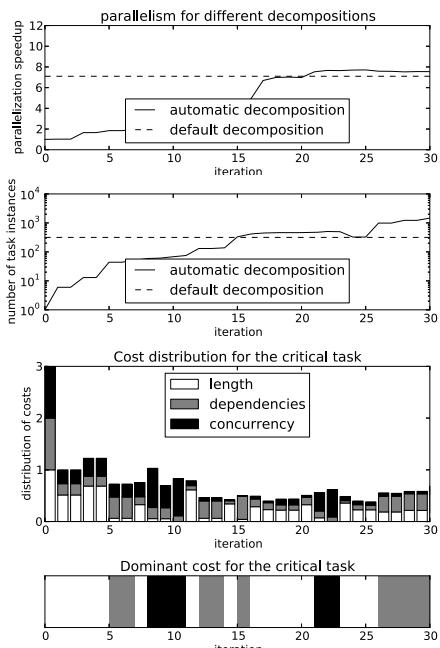


Fig. 10. Sparse LU on 8 cores

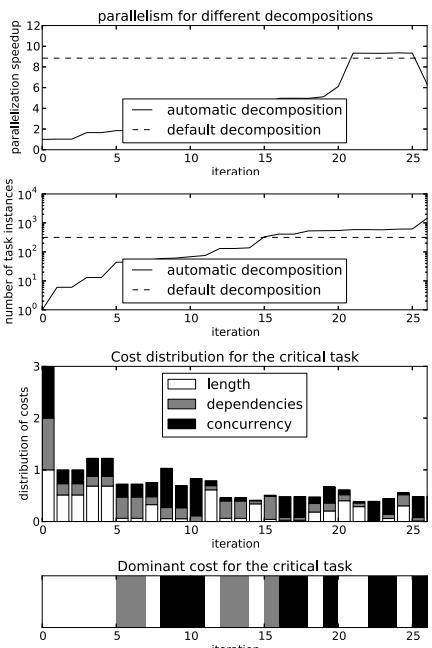


Fig. 11. Sparse LU on 16 cores

of the parallel execution of the tested decomposition. Thus, changes the normalized concurrency cost, while dependency and length cost remain the same. Figures 10 and 11 illustrate potential decompositions for Sparse LU for executing on machines with 8 and 16 cores. In the experiments with 8-core target machine (Figures 10), the reference OmpSs decomposition achieves the speedup of 7.1 at the cost of generating 316 task instances. The automatic search finds a wide range of decompositions (iterations 21-30) that provide slightly higher parallelism than the reference decomposition. On the other hand, in the experiments with 16-core target machine (Figures 11), the reference decomposition achieves the speedup of 8.85 (316 instances). The algorithm finds only five decompositions (iterations 21-25) that provide higher parallelism than the default decomposition. It is also interesting to note that in the experiment with 16-core target machine, the algorithm more often refines the decomposition using the concurrency criterion. This happens because, despite the fine granularity of decompositions, the algorithm cannot find decomposition with parallelism close to the theoretical maximum of 16 (number of cores in the target machine).

6 Discussion: Biting the Bullet of Real Workloads

This paper demonstrates that our automatic technique can find significant parallelism in a few small applications. In this section, we discuss techniques to extend scalability and applicability of our approach, and therefore allow processing realistic workloads.

Scalability of our approach concerns the execution time of the automatic search. Valgrind dynamic instrumentation presents the most computation intensive part of our technique. However, we already significantly reduced the impact of dynamic instrumentation, by implementing the workflow in which dynamic instrumentation is done only once, and then the generated logs are used offline to browse various task decompositions. Currently, we are further tackling this overhead by porting the dynamic instrumentation from Valgrind to LLVM [10]. LLVM allows us more optimized dynamic instrumentation, as well as bypassing some part of instrumentation from run-time to compile-time. Our initial studies estimate that LLVM migration could accelerate the dynamic instrumentation by a factor of 5-10 x . Also, in the part of offline decomposition exploration, we are studying various divide-and-conquer techniques. These techniques let us evaluate parallelism of smaller sections of execution (with lower number of task instances), and then combine these partial results to reconstruct the total execution.

On the other hand, applicability of our approach concerns analyzing codes that are **not parallelization friendly**. In some applications, parallelism cannot be exposed just by decomposing the application into loops and functions. In these cases, our approach first must identify memory objects whose access patterns impede automatic parallelization. The tool should also pinpoint the culprit code sections, and advice the programmer how to change problematic memory access patterns. Once the programmer changes the problematic access patterns in the sequential application, the application should become more parallelization friendly, and automatic parallelization should achieve better results.

7 Related Work

Numerous tools to assist parallelization have been proposed in the past years both from the academia and the industry. Regarding tools proposed by the academia, the ones closest to the environment that has been proposed in this paper are Embla, Kremlin, and Alchemist. In particular, Embla [11] is a Valgrind-based tool that estimates the potential speed-up for Cilk programs. On the other hand, Kremlin [8] identifies regions of a serial program that can be parallelized with OpenMP and proposes a parallelization planner for the user to parallelize the target program. Finally, Alchemist [17] identifies parts of code that are suitable for thread-level speculation. The major drawbacks of these tools are that they are limited to fork-join parallelism and that they offer very little qualitative information about the target program (no useful visualization support).

On the other side, the industry have also been recently developing their solutions for assisted parallelization. For example, Intel's Parallel Advisor [5] assists parallelization with Thread Building Blocks (TBB) [13]. Parallel Advisor provides timing profile that suggests to the programmer which loops should be parallelized. Critical Blue provides Prism [3], a tool to do “what-if” analysis that anticipates the potential benefits of parallelizing certain parts of the code. Vector Fabrics provides Pareon [7], another tool for “what-if” analysis to estimate the benefits of parallelizing loop iterations. All the three mentioned tools provide rich GUI and visualization of the potential parallelization. However, none of the tools offers automatic exploration of parallelization strategies. Moreover, they do not provide any API to automate the search for the optimal parallelization strategy as the one proposed in this paper.

8 Conclusions and Future Work

In this paper, we have proposed a technique to automate the exploration of parallelization strategies based on a task decomposition approach. We have presented an effective search algorithm that aims to find an efficient task decomposition of codes. We have defined a set of key metrics and heuristics that lead the iterative process of refining task decomposition in order to increase the parallelism of the code. The metrics collect information such as the length (duration) of the tasks, the dependencies among tasks and tasks' concurrency level. A cost function that takes into account these metrics has been proposed to guide the parallelization process. In our experiments, we demonstrate that our search algorithm is able to find task decompositions that provide enough parallelism in the application to fully utilize the target multicore processor architecture.

As future work, we have identified the need to include a new metric that evaluates the cost of expressing a decomposition using the syntax (and constraints) offered by the target parallel programming model (for example, traditional fork-join, dataflow, ...). The automatic search should be able to quantify how expressible (or viable) a decomposition could be and to use this information to guide the search. The result would be an efficient task decomposition that can be straightforwardly expressed using a specific programming model.

References

1. Benkner, S.: Vfc: The vienna fortran compiler. *Scientific Programming* 7(1), 67–81 (1999)
2. Bik, A., Girkar, M., Grey, P., Tian, X.: Efficient exploitation of parallelism on pentium iii and pentium 4 processor-based systems (2001)
3. Critical Blue. Prism, <http://www.criticalblue.com/> (active on June 27, 2013)
4. Blume, W., Doallo, R., Eigenmann, R., Grout, J., Hoeflinger, J., Lawrence, T., Lee, J., Padua, D.A., Paek, Y., Pottenger, W.M., Rauchwerger, L., Tu, P.: Parallel programming with polaris. *IEEE Computer* 29(12), 78–82 (1996)
5. Intel Corporation. Intel Parallel Advisor, <http://software.intel.com/en-us/intel-advisor-xe> (active on June 27, 2013)
6. Duran, A., Ayguadé, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X., Planas, J.: Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters* 21(2), 173–193 (2011)
7. Vector Fabrics. Pareon, <http://www.vectorfabrics.com/products> (active on June 27, 2013)
8. Garcia, S., Jeon, D., Louie, C.M., Taylor, M.B.: Kremlin: rethinking and rebooting gprof for the multicore age. In: PLDI, pp. 458–469 (2011)
9. Jost, G., Labarta, J., Gimenez, J.: Paramedir: A tool for programmable performance analysis. In: Bubak, M., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2004. LNCS, vol. 3036, pp. 466–469. Springer, Heidelberg (2004)
10. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis and transformation, San Jose, CA, USA, pp. 75–88 (March 2004)
11. Mak, J., Faxén, K.-F., Jansson, S., Mycroft, A.: Estimating and Exploiting Potential Parallelism by Source-Level Dependence Profiling. In: D’Ambra, P., Guerracino, M., Talia, D. (eds.) Euro-Par 2010, Part I. LNCS, vol. 6271, pp. 26–37. Springer, Heidelberg (2010)
12. Nethercote, N., Seward, J.: Valgrind, <http://valgrind.org/> (active on June 27, 2013)
13. Pheatt, C.: Intel threading building blocks. *J. Comput. Sci. Coll.* 23(4), 298–298 (2008)
14. Pillet, V., Labarta, J., Cortes, T., Girona, S.: PARAVER: A Tool to Visualize and Analyze Parallel Code. In: WoTUG-18 (1995)
15. Subotic, V., Ferrer, R., Sancho, J.C., Labarta, J., Valero, M.: Quantifying the Potential Task-Based Dataflow Parallelism in MPI Applications. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011, Part I. LNCS, vol. 6852, pp. 39–51. Springer, Heidelberg (2011)
16. Wilson, R.P., French, R.S., Wilson, C.S., Amarasinghe, S.P., Anderson, J.-A.M., Tjiang, S.W.K., Liao, S.-W., Tseng, C.-W., Hall, M.W., Lam, M.S., Hennessy, J.L.: Suif: An infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Notices* 29(12), 31–37 (1994)
17. Zhang, X., Navabi, A., Jagannathan, S.: Alchemist: A transparent dependence distance profiling infrastructure. In: CGO 2009 (2009)

CoreTSAR: Adaptive Worksharing for Heterogeneous Systems^{*}

Thomas R.W. Scogland¹, Wu-chun Feng¹,
Barry Rountree², and Bronis R. de Supinski²

¹ Department of Computer Science, Virginia Tech, Blacksburg, VA 24060 USA
² Center for Applied Scientific Computing, Lawrence Livermore National Laboratory,
Livermore, CA 94551 USA

Abstract. The popularity of heterogeneous computing continues to increase rapidly due to the high peak performance, favorable energy efficiency, and comparatively low cost of accelerators. However, heterogeneous programming models still lack the flexibility of their CPU-only counterparts. Accelerated OpenMP models, including OpenMP 4.0 and OpenACC, ease the migration of code from CPUs to GPUs but lack much of OpenMP’s flexibility: OpenMP applications can run on any number of CPUs without extra user effort, but GPU implementations do not offer similar adaptive worksharing across GPUs in a node, nor do they employ a mix of CPUs and GPUs. To address these shortcomings, we present CoreTSAR, our library for scheduling cores via a task-size adapting runtime system by supporting worksharing of loop nests across arbitrary heterogeneous resources. Beyond scheduling the computational load across devices, CoreTSAR includes a memory-management system that operates based on task association, enabling the runtime to dynamically manage memory movement and task granularity. Our evaluation shows that CoreTSAR can provide nearly linear scaling to four GPUs and all cores in a node *without* modifying the code within the parallel region. Furthermore, CoreTSAR provides portable performance across a variety of system configurations.

1 Introduction

Heterogeneity is becoming more prevalent in all areas of computing, from supercomputers to cell phones. The increasing prevalence of GPUs and computational coprocessors has spawned a vast array of tools and programming models over the past several years, but the majority of codes remain CPU-only. Of these new models, Accelerated OpenMP holds the promise of increasing adoption, especially in high-performance computing (HPC). Accelerated OpenMP models, including OpenACC [6] and OpenMP 4.0 [5], extend the classic OpenMP

* This work was supported in part by the Air Force Office of Scientific Research (AFOSR) Computational Mathematics Program via Grant No. FA9550-12-1-0442, NSF I/UCRC IIP-1266245 via the NSF Center for High-Performance Reconfigurable Computing (CHREC) and a DoD National Defense Science & Engineering Graduate Fellowship (NDSEG).

preprocessor directives with the capability to offload data-parallel regions to accelerators¹.

Assuming the ease of use is sufficient to convince developers to embrace accelerators, one major issue still remains. Accelerated OpenMP only offloads computation to a single device (e.g., GPU), leaving the CPU cores or other accelerators in the system idle. There are mechanisms that allow users to manually partition their work across devices but *no* direct support for cross-device worksharing. As a result, users must *manually* partition their job, manage memory coherence, and load balance across devices. In the best case, a user may use a task scheduling library such as OmpSs or StarPU to handle load balancing and memory transfers. Even then, however, the user must manually divide their workload into explicit tasks.

Our solution, CoreTSAR (short for Core Task-Size Adapting Runtime), provides a cross-device worksharing construct for Accelerated OpenMP. The key extension to make this possible is a memory-management approach that allows a user to specify the association between their computation and data, that frees CoreTSAR to handle coherence and task granularity automatically. CoreTSAR includes a proposed set of extensions to Accelerated OpenMP, the design of a cross-device memory manager, scheduling policies to support the clauses, and a real-world implementation of the system. Together, they provide adaptive worksharing of parallel loop regions across an arbitrary number of devices with an arbitrary number of distinct address spaces. In this paper, we make the following contributions:

- Accelerated OpenMP extensions to adaptively workshare parallel regions across an arbitrary number of arbitrarily heterogeneous devices
- The design and implementation of a task-associative, memory-management interface, thus allowing CoreTSAR to adapt task granularity at runtime
- A library and C API implementation of our scheduler and memory manager
- An evaluation demonstrating that CoreTSAR can improve performance over existing task-management approaches.

The remainder of the paper is composed as follows. Section 2 provides background and motivation. Section 3 describes the design and implementation of CoreTSAR, including our task-management concept, scheduling mechanisms, and memory management. Section 4 presents our evaluation. Finally, we finish with related work in Section 5 and conclusion in Section 6.

2 Background and Motivation

Models for heterogeneous computing generally fall into three categories: offload models; block and grid models; and task block models. Domain specific languages are also available, but we focus on general purpose options. Each of the

¹ Since our evaluation is based on GPU accelerators, the terms “accelerator” and “GPU” are used interchangeably throughout.

```

void kmeans_it(int *m, float *fo, float *fc,
               size_t no, size_t nco, size_t ncl) {
//OpenMP
#pragma omp parallel for
//Accelerated OpenMP
#pragma acc parallel for copyout(m[0:no]) \
    copyin(fc[0:nco*ncl],fo[0:no*nco])
//Accelerated OpenMP + extension
#pragma acc parallel for pcopy(m[1:no]) \
    copyin(fc[0:nco*ncl]) copyin(fo[no*nco]) \
    hetero(i, all, adaptive, default, 10)
    for (i=0; i<no; i++) {
        m[i] = findc(no,ncl,nco,fo,fc,i);
    }
}

__global__ void
void it_gpu(int *m, float *fo, float *fc,
            size_t no, size_t nco, size_t ncl,
            size_t start, size_t end) {
    uint i = blockIdx.x * blockDim.x + threadIdx.x;
    if(i < end-start) {
        m[i] = findcu(no,ncl,nco,fo,fc,i+start);
    }
}

void kmeans_it(int *m, float *fo, float *fc,
               size_t no, size_t nco, size_t ncl,
               size_t start, size_t end) {
dim3 dB, dG;
dB.x = 64;
dB.y = dB.z = 1;
dG.x = ((end-start)/dB.x)+1;
dG.y = dG.z = 1;
cudaMalloc(&cm, no);
cudaMalloc(&cfo, no*nco);
cudaMalloc(&cfc, no*ncl);
cudaMemcpy(cm, m, no, cudaMemcpyHostToDevice);
cudaMemcpy(cfo, fo, no*nco, cudaMemcpyHostToDevice);
cudaMemcpy(cfc, fc, no*ncl, cudaMemcpyHostToDevice);
it_gpu(<<<dG,dB>>>(
    cm, cfo, cfc, no, nco, ncl, 0, no);
cudaMemcpy(m, cm, no, cudaMemcpyDeviceToHost);
}
}

#pragma omp target device(smp,cuda)
void it_helper(int *m, float *fo, float *fc,
               size_t no, size_t nco, size_t ncl,
               size_t start, size_t end);

#pragma omp target device(cuda)
__global__ void
void it_gpu(int *m, float *fo, float *fc,
            size_t no, size_t nco, size_t ncl,
            size_t start, size_t end);
/* implementation same as CUDA, see left */

#pragma omp target device(smp)
#pragma omp task input([no*nco]fo, [ncl*ncl]fc)\ 
    inout([end-start]m)
void it_helper(int *m, float *fo, float *fc,
               size_t no, size_t nco, size_t ncl,
               size_t start, size_t end){
    for (int i=0; i<end-start; i++) {
        m[i] = findc(no,ncl,nco,fo,fc,i+start);
    }
}

#pragma omp target device(cuda) copy_deps\ 
    implements(it_helper)
#pragma omp task input([no*nco]fo, [ncl*ncl]fc)\ 
    inout([end-start]m)
void it_cuhelper(int *m, float *fo, float *fc,
                  size_t no, size_t nco, size_t ncl,
                  size_t start, size_t end) {
    dim3 dB, dG; dB.x = 64; dB.y = dB.z = 1;
    dG.x = ((end-start)/dB.x)+1; dG.y = dG.z = 1;
    it_gpu(<<<dG,dB>>>(
        cm,cfo,cfc,no,nco,ncl,start,end));
}

void kmeans_it(int *m, float *fo, float *fc,
               size_t no, size_t nco, size_t ncl) {
const int BS = 1000;
for (i=0; i<no-BS; i+=BS) {
    it_helper(&(m[i]), fo, fc, no, nco, ncl, i, i+BS);
}
    it_helper(&(m[i]), fo, fc, no, nco, ncl, i, no);
#pragma omp taskwait
}
}

```

Fig. 1. A basic kmeans kernel as implemented in OpenMP variants (top left), CUDA (bottom left) and OmpSs (right)

approaches has strengths and weaknesses in terms of programmability, performance, and flexibility, which we will discuss here in terms of the three example implementations of the kmeans clustering algorithm presented in Figure 1.

Offload models execute annotated parallel regions on a GPU. Examples include Accelerated OpenMP (OpenMP 4.0 [12], OpenACC [6], Intel OpenMP for MIC and Cray’s accelerated OpenMP [5]) and C++ AMP. Generally they require the least change from the original serial code. The upper left of Figure 1, despite its size, actually contains four different versions. If none of the pragmas are honored, then the loop executes in serial. If the OpenMP pragma on line 4 is honored, the loop is work-shared across CPU cores. To target a GPU, the OpenACC directive on lines 6 and 7 copies both the *fc* and *fo* arrays to the GPU and *m* to and from it. This directive *moves* the loop from the CPU to a GPU: when the **parallel for** directive is applied, no CPU cores (or other GPUs) participate in the loop. One must manually split the loop and data to target multiple devices.

Block and grid models represent data-parallel kernels as a grid of blocks of threads, where synchronization in a kernel is only possible within the blocks. OpenCL [11] and NVIDIA’s CUDA both fall into this group. They are some of

the most efficient and most used GPU programming models, offering low-level control at the cost of verbosity. The bottom left code block in Figure 1 presents a basic CUDA port of the OpenMP kmeans code. Even with the larger number of management lines, the CUDA version does not free the GPU memory, check error codes, or perform GPU selection and initialization (so the conservative default queue is used). As with the OpenMP version, this code only uses one GPU.

Task block models do not actually specify a programming model for accelerators but rather provide task scheduling across heterogeneous resources. This group includes StarPU [2] and OmpSs [9]. The right side of Figure 1 presents an OmpSs implementation of the kmeans loop. The first four pragmas specify that the `it_helper` function is an OmpSs task that depends on `fo`, `fc` and a slice of `m`. The `implements` clause on line 20 informs OmpSs that `it_cudahelper`, which uses the CUDA implementation at the bottom left minus the memory movement, is the CUDA implementation of `it_helper`. In `kmeans_it()` the tasks are enqueued by calling the helper function with appropriately offset pointers. OmpSs automatically distributes this work across all CPU cores and GPUs. This flexibility provides performance portability, at a cost. The user must partition their work into tasks appropriate for either an entire GPU or a single CPU core simultaneously.

These models are all useful under the right circumstances. Optimally we would have the fine-grained control of block and grid models, the simplicity and programmability of offload models and the runtime flexibility and performance portability of task block models. We propose syntax and offer a runtime implementation of CoreTSAR(Task-Size Adapting Runtime), a system that can be used to add flexibility to offload and block and grid models, and also to extend task-block models with adaptive task granularity.

3 Design and Implementation

Our primary goal with CoreTSAR is to create a runtime to support worksharing across devices for use with Accelerated OpenMP. This goal imposes certain design constraints. Most importantly, it must not require any changes, even for memory movement, to the loop body beyond those for Accelerated OpenMP. For example, no pragmas or API calls may be inserted into the loop, nor memory access patterns be changed, as task scheduling systems often require. All information necessary for CoreTSAR to provide the correct data for *any* range of iterations to *any* device’s memory space must be provided in the directive outside the loop. Further, we must preserve data consistency outside the region: main memory must hold the same values when the loop exits as it would have with Accelerated OpenMP.

Figure 2 depicts the syntax of our extension. The `hetero()` clause specifies how to treat the region. Specifically, whether to schedule it, which scheduler to use, which devices to schedule it across and, if desired, initial values for the work-split ratio. The `pcopy()` clause specifies the association between iterations and data. It is similar to a copy clause in OpenACC, except that it only applies to

loop directives and only copies the specific memory elements that are associated with the range of iterations assigned to a given device. We support current OpenACC `copyin` and `alloc` data clauses in order to replicate the entire input or allocation, `copyout` is unsupported. The third pragma in the top left block of Figure 1 uses our extension to divide the loop across all devices using the adaptive scheduler, copying all of `fc` and `fo` as input, but only the necessary elements of `m` as output.

```
//items in {} are optional
#pragma acc parallel for hetero(<condition>{,<devices>{,<scheduler>{,<ratio>{,<div>{}}}}})\n
    pcopy{in/out}(<var>[<cond>:<num>{:<boundary>}]) persist(<var>)
#pragma acc depersist(<var>)
```

Fig. 2. Our proposed extension

The remainder of this section discusses the design and implementation of a runtime system to support the proposed extension. Our design has two main components: the scheduling and task inference portion; and the memory specification and management portion.

3.1 Scheduling and Tasks

As our previous work in heterogeneous task scheduling showed [16], adaptive worksharing by predicting performance, rather than employing work-queues or discrete tasks, can significantly reduce overhead in heterogeneous scheduling. Overhead is reduced by assigning work *statically* within a region, and re-balancing on the next entry into the region, thus reducing the number of tasks to manage and launch. The significant downside to our previous approach however was that the model only supported the modeling of performance across two devices, preventing it from targeting systems with higher levels of heterogeneity. That work also proposed a solution as future work, using an integer-based optimization approach.

The original integer-based optimization directly computed the number of iterations to assign to each device, but incurs too much overhead to be used for runtime scheduling, on the order of seconds for less than ten devices. Figure 3 presents an alternative approach designed for CoreTSAR, optimized for use at runtime. Our new linear optimization minimizes the total deviation between the predicted runtimes for all devices to run their assigned work, and expresses output as the fraction of total iterations that should be assigned to a given device. Switching to fractional output removes the costly refinement to integer output and increases numerical stability. The downside is that fractional output allows up to n iterations to go unassigned, any iterations left-over this way are assigned in round-robin fashion to devices in descending order of performance.

This design assumes that trading one CPU core to control a GPU will improve performance. However, some applications benefit more from the CPU core. Thus, CoreTSAR also includes a heuristic allowing it to convert a GPU controlling thread back to a CPU thread in such cases. After each pass the time per

$$\min\left(\sum_{j=1}^{n-1} t_j^+ + t_j^-\right) \quad (2)$$

$$\sum_{j=0}^n f_j = 1 \quad (3)$$

$$f_2 * p_2 - f_1 * p_1 = t_1^+ - t_1^- \quad (4)$$

$$f_3 * p_3 - f_1 * p_1 = t_2^+ - t_2^- \quad (5)$$

$$\vdots$$

$$f_n * p_n - f_1 * p_1 = t_{n-1}^+ - t_{n-1}^- \quad (6)$$

Fig. 3. Linear program variables, objective and constraints

iteration (TPI) of each GPU is compared against that of the slowest CPU core. If a GPU's TPI falls below that value for two consecutive iterations, the GPU thread releases the GPU and continues as a CPU thread.

3.2 Static Scheduling

On the first entry into a region, our static schedule uses the linear program to assign iterations, then reuses that result for all subsequent passes. To increase portability, we compute default relative times per iteration at runtime rather than using a precomputed static value (the user can also specify a value). Our default assumes that one instruction cycle on a GPU core, or SIMD lane on a multiprocessor, takes the same time as one cycle on a single SIMD lane of a CPU core. We thus compute the time per iteration for each GPU as $p_g = \frac{1}{m/s}$ and for CPU cores as $1 - p_g$ (where m is the number of cores on the GPU and s the SIMD width of a CPU core; in the case of multiple GPUs, all devices are normalized to the largest). For applications that are not dominated by floating-point computation, we have considered models that include several other factors, including memory bandwidth and integer performance, but leave these for future work.

3.3 Adaptive Scheduling

Our adaptive schedules (*Adaptive*, *Split* and *Quick*) use the static schedule for the first pass. We then use the time that each device takes to complete its iterations in the preceding pass, kept as a weighted average over up to five passes, as input to our linear program for the next pass. All recurring overheads required to execute an iteration on a particular device are included in that time (but not one-time overheads such as the copying of persistent data). Thus, we incorporate data-movement and launch overheads into the cost of each iteration and naturally account for them. The *Adaptive* schedule trains on the first instance of the region and then each subsequent instance. The *Split* schedule breaks each region into several evenly split sub-regions based on the *div* input.

Each sub-region is then treated individually and scheduled as Adaptive would a full region, providing faster load balancing at the cost of increased overhead. The *Quick* schedule balances between the *Split* and *Adaptive* schedules by executing a small sub-region for its first training phase, similar to the way *Split* starts. It then immediately schedules all remaining iterations of the first region instance and uses the *Adaptive* schedule for any subsequent instances. This schedule suits applications that cannot tolerate a full instance using the static schedule or the overhead of extra scheduling steps in every pass.

3.4 Memory Management

Efficient and minimal data movement is essential to the performance of heterogeneous codes. To handle memory movement without explicit tasks, we allow the user to specify the association between a loop range and its input and output pattern. Our interface takes a pointer, optionally the size of each element and for each dimension whether to associate that dimension with the iterator, the length, and the number of boundary values required. If a dimension is marked as an iterator dimension, then all values in that dimension corresponding to assigned iterations are copied. For example, Figure 4 shows the data associations for two simple cases. On top, the `pcopy(mat[false:10][true:10])` clause specifies that a 10×10 matrix is to be managed and the iterator will select the column, since the column dimension's condition is true. The second example uses `pcopy(mat[true:10:1][false:10])` instead, which has a drastic effect on the result. Now, the rows are associated with the iterator rather than the columns, since the association value is true for the row dimension rather than the column dimension. Further, this example is designed for a stencil-type code which requires boundary values as input, but not as output, so the boundary size in the row dimension has been set to one to copy one row above and one below as input.

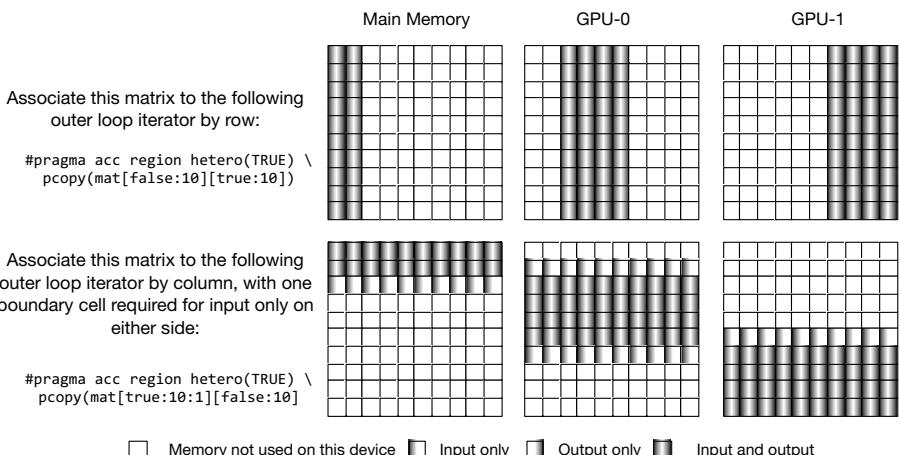


Fig. 4. Example memory associations, assuming a pass in which two iterations are assigned to the CPU device, and four each to two GPUs

Our design handles reductions by doing partial reductions on each device, and a finalizing pass on the CPU into the final target variable. The high-level interface has no extension for this, OpenMP syntax is already sufficient, but our library API provides a mechanism similar to that of user-defined reductions to manually construct more complex reductions. While this interface does not support fully general input and output, we believe it to be a worthwhile first step in that direction, one we intend to pursue further in future work.

4 Results and Evaluation

To evaluate our prototype, we have ported six applications to OpenACC directives and extended them with CoreTSAR scheduling. We evaluate these applications across a range of systems, schedulers, and configurations. We also provide a performance comparison of three of the applications with OmpSs and StarPU.

4.1 Benchmarks

Our six benchmarks exhibit a range of characteristics that reflect possible use cases. Minimal changes were made to port the original serial or OpenMP version of each benchmark to OpenACC and subsequently CoreTSAR.

The values in Table 1 characterize the benchmarks. The number of passes, relative pass length, number of iterations per pass and ratio particularly impact CoreTSAR operation. We derive the CPU/GPU ratio, the relative performance of the CPU cores of our test system as compared to one of its GPUs, from the best found through brute-force testing. A score of 1.0 allocates all iterations to the CPU cores, a score of 0 allocates all iterations to the GPU.

First, CG is the conjugate gradient benchmark from the NAS parallel benchmarks OMP version 3.3. The offloaded portion runs many (1,900) short passes and contains a high percentage of indirect array accesses. This version of CG is relatively unsuitable for GPUs with a ratio of 0.85, implying that a GPU is only 20% faster than one CPU core. CORR and GEMM are both from the PolyBench/GPU [10] suite. CORR is an upper triangular matrix solver with a completely unbalanced workload. Iteration i computes $n - i$ output values so

Table 1. Benchmark characteristics, times in seconds (* polybench suite benchmarks can have a variable number of passes, we use 10 for our tests)

Benchmark	Passes	Iterations/ pass	Time/pass (CPU)	GPU runtime	Resulting CPU/GPU Ratio
CG	1900	75,000	0.02	273.04	0.85
CORR	10*	2,048	6.36	70.97	0.01
GEM	1	258,800	1098.10	107.43	0.06
GEMM	10*	2,048	1.262	3.04	0.01
Helmholtz	100	4,000	0.08	73.64	1.00
kmeans	7	1,210,000	1.14	4.79	0.41

no two iterations, or ranges of iterations, are the same. GEMM is a general matrix multiplication that we schedule row-wise, it is highly suitable to GPU computation but is sensitive to memory contention and NUMA effects. GEM is a molecular modeling visualization application that computes the electrostatic potential along the surface of a macromolecule. While GEM is extremely well suited to GPU computation, as we have shown in our previous work [1,7], it only runs a single pass by default. Helmholtz implements the Helmholtz equation using the Jacobi method. Our implementation is based on a CPU OpenMP implementation that is not well suited to GPUs due to its memory access pattern and conditional nature. Kmeans implements the kmeans clustering algorithm, and is near an even ratio (0.5): running on a GPU takes the same amount of time as on the CPU cores. It also has many iterations per pass, which allows CoreTSAR to make very fine grained adjustments to each device’s workload.

Overall, the benchmarks exhibit a wide range in each category. Pass counts for example range from one to 1,900; iterations per pass from 2,048 to 1,210,000; and ratios from 0.01 to 1.0.

4.2 Experimental Setup

CoreTSAR has been implemented in full as a C library, with a source-to-source translator implementing the pragma syntax based on python and libclang, on top of PGI OpenMP and OpenACC. To solve our linear optimization problem, we use the lp_solve library[4], an optimized linear program solver, in a mode that allows it to incrementally refine the solved tableau on each pass. We evaluate all five systems listed in Table 2. However, unless otherwise specified, we run tests on escafowne. Each machine has the same OS configuration (Debian Squeeze).

Unless otherwise noted, all benchmarks are implemented with OpenACC directives and compiled with the PGI compiler version 12.9. OmpSs tests are compiled with mvncc version 1.3.5.8, using the performance configuration of the NANOS++ libraries and the versioning-stack scheduler. OmpSs options for prefetching and asynchronous transfers are used only on helmholtz, the other two benchmarks incur a slowdown when they are used. The modified CoreTSAR and StarPU versions compared with OmpSs are compiled with nvcc and linked with GNU OpenMP (gomp). The StarPU implementations used the “dmda” scheduler with the history based performance model, trained on ten plus runs before results were collected. CUDA toolkit version 4.1 is used in all tests. All threads

Table 2. Hardware composition of each test platform, Intel CPUs and NVIDIA GPUs throughout

System name	CPU Model	CPU Cores/die	CPU Dies	CPU RAM	GPU Model	GPU Cards	GPU Cores	GPU RAM
amdlow3	E3300	2	1	2,012MB	Tesla C2050	1	448	3GB
armor1	E5405	4	2	3,964MB	GeForce GT 520	1	48	1GB
dna2	i5-2400	4	1	7,923MB	GeForce GTX 280	1	240	1GB
escafowne	X5550	4	2	24,154MB	Tesla C2070	4	448	6GB

are bound to cores at the beginning of execution, before memory allocation and initialization.

All reported performance measurements time the core phase of the benchmark only. All marshaling, staging, copying, or other preparation necessary to use an implementation is also included. We exclude only identical application parts such as file IO and result verification.

4.3 CoreTSAR Performance

Figure 5 presents the performance impact of CoreTSAR. The graph represents speedup over a statically scheduled eight core CPU run, with a black bar marking the baseline in each sub-plot. Without CoreTSAR unmodified code would be either at that baseline, or the GPU mark with one GPU. All others, including static and GPU for a GPU count greater than one, use CoreTSAR’s facilities. Of the six benchmarks, four scale nearly linearly from one to four GPUs given the right scheduler. As expected, Helmholtz and CG do not.

Both CORR and GEMM display high GPU suitability, each approaching an overall speedup of $250\times$ on four GPUs. We obtain the best performance for GEMM on escaflowne by statically scheduling the computation; the quick scheduler is almost as good. CORR is less predictable. Its heterogeneous workload per iteration causes the adaptive, split, and quick schedulers all to make incorrect early decisions about how to assign work. While overall the static schedule fares best, the split schedule overtakes it for four GPUs because the linear model stops

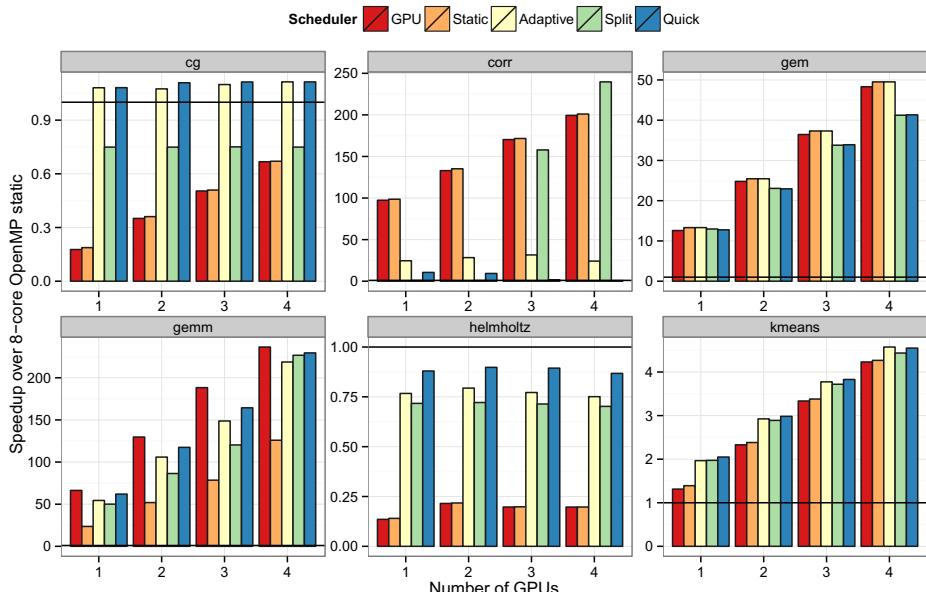


Fig. 5. Performance of CoreTSAR enabled benchmarks on escaflowne normalized to eight core OpenMP

assigning work to the *CPU* cores and adaptively schedules work across the four GPUs. It is worth noting that both the adaptive and quick schedulers converge on efficient assignments after the first few passes, but the cost of the early mispredictions overtake that benefit. Using the ratios from a previous run produces even more favorable results, we intend to investigate this further in future work.

Both GEM and kmeans also scale near linearly to four GPUs, but with slightly different characteristics. GEM benefits most from the default static split across CPUs and GPUs. Its natural ratio is so close to the default split that, while the dynamic schedulers can improve on it, the overhead of synchronization and additional assignments increases the overall execution time. On the other hand, kmeans does best with the adaptive schedules. The optimal choice shifts between quick and adaptive depending on the number of GPUs, but they remain within 5% of each other regardless. Unlike GEM, kmeans is more CPU suitable, and the GPU or static schedules underutilize those resources.

CG is not GPU averse, since it can benefit from the use of a GPU, but it does not scale to more than one GPU. When we allow use of more than one GPU, the increased memory transfer overhead causes a slowdown. This example demonstrates that some applications can benefit from GPUs, but may still need to back off of their use. Helmholtz, on the other hand, gains no benefit. It does however stay within 20% of the CPU performance given a scheduler that can quickly deactivate GPUs. When allowed to use cross-run historical data, Helmholtz consistently matches the CPU performance.

Overall, our results demonstrate that CoreTSAR adapts well to different workloads. We attain good scaling for applications that are amenable to GPU computing or coscheduling. Alternatively, CoreTSAR backs off appropriately for those that are not.

4.4 Adaptation across Machines

We have shown that CoreTSAR can provide efficient worksharing across a varied number of GPUs in one system. We now evaluate its performance portability across a range of systems, those listed earlier in Table 2. Of these systems, escafowne, used for our primary evaluation, is the largest and the only multi-gpu system. Representing CPU-centric systems, armor1 contains two capable quad-core Intel Xeon CPUs with a low-power consumer desktop GPU. Conversely amdlow3 contains a dual-core Celeron CPU and a powerful Tesla C2050 GPU. Lastly dna2 represents a more typical workstation with a mid-range quad core CPU and previous-generation consumer GPU.

Figure 6a presents results for the three benchmarks that benefit from coscheduling. Helmholtz is CPU-suited on all machines, and corr and gemm are GPU-suited on all machines, so we elide their results for space. Each result is normalized to the best performing configuration for that benchmark on that system with the best time at 1.0 and lower being worse. First, while CG gains minimally in performance on escafowne, and not at all on armor1 and dna2, it attains a 2× speedup over the CPU result on amdlow3. Second, we find that appropriate adaptive schedulers tend to hold across different systems. For both

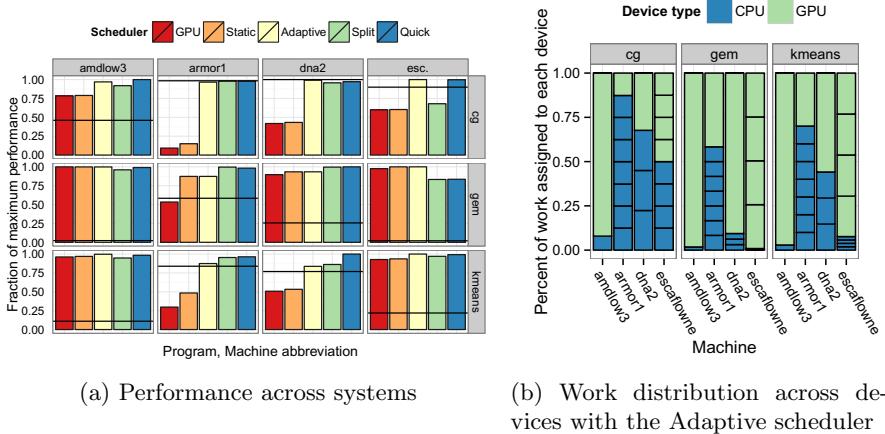


Fig. 6. Cross-system performance and assignment results. The black bar in (a) represents OpenMP CPU performance across all cores in that system.

CG and kmeans the quick scheduler performs best across all tested machines, regardless of their composition. GEM is a bit different, in that on some systems the quick or split schedulers are slightly beneficial, but on escaflowne they are slightly worse than the static split. Run in production, where a user will invoke the visualization routine repeatedly, the quick scheduler is fastest everywhere.

Even though the appropriate scheduler remains the same across machines, the workload distribution across resources shifts significantly, as Figure 6b shows. Each bar represents the total amount of work run during the course of a given benchmark, each CPU cell represents the work in iterations completed by a particular CPU core, and each GPU cell the work completed by a particular GPU. Each benchmark’s distribution shifts across machines based on the machine’s suitability. The most striking of these is CG, which uses the GPU for very little on three of the four systems, but almost exclusively on amdlow3.

4.5 Comparison with State of the Art

Our evaluation has demonstrated that CoreTSAR achieves good and portable performance. We now compare its performance to that of two state-of-the-art heterogeneous task schedulers, OmpSs and StarPU. In order to compare all three fairly, we ported three of our benchmarks (kmeans, Helmholtz and GEMM) from OpenACC to CUDA/C and extended that version with each scheduler. The CoreTSAR code evaluated here uses the exact same CUDA/C implementations as OmpSs and StarPU, in fact *linked from the same binary*. Additionally, we configure all three schedulers to use the same initial granularity for each benchmark, amounting to approximately 2000 tasks per pass.

Figure 7 presents our results for these benchmarks on escaflowne for all schedulers targeting all eight CPU cores and four GPUs. The result for GEMM is rather unexpected, with OmpSs and StarPU both about 2 \times slower than the

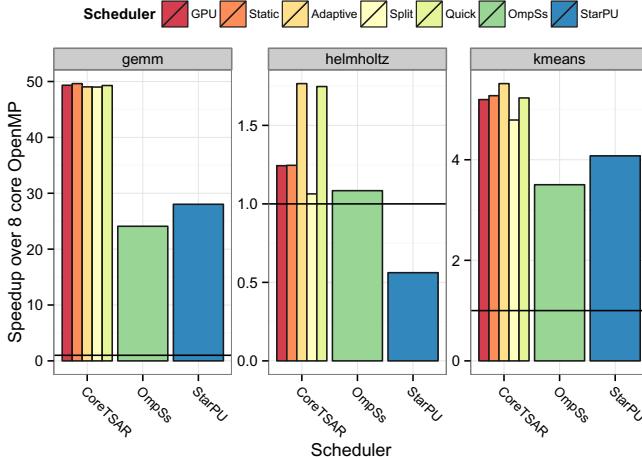


Fig. 7. Speedup comparison between CoreTSAR, OmpSs and StarPU

CoreTSAR version. We initially suspected that this was the result of extra data transfers, or even an error in our implementation of the memory movement in OmpSS and StarPU, but manually minimizing the data transfers did not materially change the result. Rather, overhead from creation, management and scheduling of individual tasks is to blame for the difference. CoreTSAR has the advantage of automatically altering the granularity of tasks, rather than running user-defined chunks. In the adaptive scheduler for example, a single kernel is run on each GPU, whereas in OmpSs hundreds to thousands may be run. No matter how efficient the runtime, there is a cost for such fine-grained management.

Helmholtz is of interest for a different reason. The CUDA and serial C version gets materially faster by running on four GPUs, especially with the CoreTSAR adaptive or quick schedules with a nearly 75% improvement. The reason it performs so differently from the version evaluated above is that the nvcc compiler produces significantly slower CPU code than the PGI compiler used elsewhere, allowing the GPUs to outperform the CPUs. This result reinforces the idea that allowing automatic coscheduling, even in cases where it does benefit some machines or configurations, can be beneficial. OmpSs also improves on the CPU-only performance by about 5%, and StarPU almost 2× slower. Our evaluation found that this is due to underutilizing the CPU cores in favor of the GPUs, as well as the same overheads that plague GEMM.

Finally the kmeans results show OmpSs and StarPU scaling to a respectable 3.5×, but still trailing the CoreTSAR adaptive schedulers. For cases where the application launches and immediately waits for a range of related tasks, CoreTSAR consistently performs well. OmpSs and StarPU on the other hand perform well with many asynchronously launched tasks which wait rarely, and suffer when the full synchronization is more frequent. We believe these are complementary designs, and will investigate granularity adaptation for general task graph scheduling systems as future work.

5 Related Work

Task scheduling as a mechanism for easing parallel programming has a long history. Traditional applications of the approach include dynamic loop parallelization in OpenMP [8] and Intel’s TBB [15]. These mechanisms tend to offer simplified syntax for shared memory parallelism, but little to no support for heterogeneous architectures or distributed memory. Scheduling policies for these mechanisms have also been the focus of significant research. Work by Ayguadé et al. [3], directly influenced the design of CoreTSAR. They investigated the removal or extension of OpenMP schedule clauses by calculating the distribution of work in future passes through a region based on times seen for each core in previous ones. Their results showed the method was not always optimal, but that the solution was efficient and stable. Our ratio-based design works similarly, although with a different mechanism to determine the split between devices.

Task block models, such as StarPU [2] and OmpSs [9], began to tackle the problem of scheduling tasks across heterogeneous resources based on a directed acyclic graph. These models provide a powerful platform for scheduling on heterogeneous systems, but require the user to determine task granularity manually. We believe these designs to be complementary to our own, combining the ability to adaptively adjust task granularity with task-block style arbitrary graph execution could yield powerful results.

Ravi et al. [13] presented a scheduling framework for multicore systems with a single GPU that builds on their generalized reduction framework and code generator [14]. While we avoid the chunk scheduling scheme and its additional transfer overheads, they present an approach that uses chunk-based scheduling while mitigating the overhead through runtime techniques.

6 Conclusion

We present four major contributions: the design of our task-size adapting runtime for adaptive worksharing across heterogeneous devices; the design of a task-associative memory model; an implementation of the designs; and our evaluation across six codes, and four systems. Our system yields linear speedups on up to four GPUs for amenable benchmarks, and show a high degree of performance portability across a set of highly disparate system configurations. These results clearly motivate the addition of a co-scheduling interface, such as the `hetero()` clause that we propose, to Accelerated OpenMP; the results also highlight the benefits of automatically adapting task granularity at runtime.

The memory management interface that we present is the first step towards a general interface for declaring the relationship between tasks and the portions of inputs and outputs that they require. Given that information many schedulers, including ours, could automatically manage input and output, providing significant value especially as computers become more complex. We intend to investigate this expanded interface as future work.

References

1. Anandakrishnan, R., Scogland, T.R.W., Fenley, A.T., Gordon, J.C., Feng, W.-c., Onufriev, A.V.: Accelerating Electrostatic Surface Potential Calculation with Multi-Scale Approximation on Graphics Processing Units. *Journal of Molecular Graphics and Modelling* 28(8), 904–910 (2009)
2. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.-A.: STARPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 863–874. Springer, Heidelberg (2009)
3. Ayguadé, E., Blainey, B., Duran, A., Labarta, J., Martínez, F., Martorell, X., Silvera, R.: Is the *Schedule* Clause Really Necessary in OpenMP? In: Voss, M.J. (ed.) WOMPAT 2003. LNCS, vol. 2716, pp. 147–160. Springer, Heidelberg (2003)
4. Berkelaar, M., Notebaert, P., Eikland, K.: lp_solve(mixed integer) linear programming problem solver (2003), <http://lpsolve.sourceforge.net/5.0/>
5. Beyer, J.C., Stotzer, E.J., Hart, A., de Supinski, B.R.: OpenMP for accelerators. In: Chapman, B.M., Gropp, W.D., Kumaran, K., Müller, M.S. (eds.) IWOMP 2011. LNCS, vol. 6665, pp. 108–121. Springer, Heidelberg (2011)
6. CAPS Enterprise, Cray Inc., NVIDIA and the Portland Group. The openacc application programming interface, v1.0. (November 2011), <http://www.openacc-standard.org>
7. Daga, M., Scogland, T., Feng, W.: Architecture-aware mapping and optimization on a 1600-core gpu. In: 2011 IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS), pp. 316–323. IEEE (2011)
8. Dagum, L., Menon, R.: OpenMP: An Industry Standard API for Shared-Memory Programming. *IEEE Computational Science & Engineering* 5(1), 46–55 (1998)
9. Duran, A., Ayguade, E., Badia, R., Labarta, J., Martinell, L., Martorell, X., Planas, J.: OmpSs: A Proposal for Programming Heterogeneous Multi-Core Architectures. *Parallel Processing Letters* 21(2), 173–193 (2011)
10. Grauer-Gray, S., Xu, L., Searles, R., Ayalasomayajula, S.: Auto-tuning a High-Level Language Targeted to GPU Codes. cis.udel.edu
11. Munshi, A.: Khronos OpenCL Working Group and others. The opencl specification (2008)
12. OpenMP Architecture Review Board. OpenMP application program interface version 4.0 (2013)
13. Ravi, V.T., Agrawal, G.: A dynamic scheduling framework for emerging heterogeneous systems. In: 2011 18th International Conference on High Performance Computing (HiPC), pp. 1–10 (2011)
14. Ravi, V.T., Ma, W., Chiu, D., Agrawal, G.: Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations. In: ICS 2010: Proceedings of the 24th ACM International Conference on Supercomputing, ACM Request Permissions (June 2010)
15. Reinders, J.: Intel Threading Building Blocks (2007)
16. Scogland, T.R.W., Rountree, B., Feng, W.-c., de Supinski, B.R.: Heterogeneous Task Scheduling for Accelerated OpenMP. In: 2012 IEEE International Parallel & Distributed Processing Symposium (IPDPS), Shanghai, China (2012)

History-Based Predictive Instruction Window Weighting for SMT Processors

Gurhan Kucuk, Gamze Uslu, and Cagri Yesil

Yeditepe University, Faculty of Engineering and Architecture,
Department of Computer Engineering, Istanbul, Turkey
`{gkucuk, guslu, cyesil}@cse.yeditepe.edu.tr`

Abstract. In a Simultaneous Multi-Threaded (SMT) processor environment, threads share datapath resources, and resource allocation policy directly affects the throughput metric. As a way of explicit resource management, resource requirements of threads are estimated based on several runtime statistics, such as cache miss counts, Issue Queue usage and efficiency metrics. Controlling processor resources indirectly by means of a fetch policy is also targeted in many recent studies. A successful technique, Speculative Instruction Window Weighting (SIWW), which speculates the weights of instructions in Issue Queue to indirectly manage SMT resource usage, is recently proposed. SIWW promises better performance results compared to the well-accepted ICOUNT fetch policy. In this study, we propose an alternative fetch policy that implements SIWW-like logic using a history-based prediction mechanism, History-based Predictive Instruction Window Weighting (HPIWW), avoiding any types of speculation hardware and its inherent complexity. As a result, we show that HPIWW outperforms SIWW by 3% on the average across all simulated workloads, and dissipates 2.5 times less power than its rival.

Keywords: Simultaneous Multi-Threaded processors, Resource Management.

1 Introduction

Simultaneous Multi-Threaded (SMT) processors are very useful for utilizing processor resources in both time and space domain. In a typical superscalar processor, only one thread is allowed to run its instructions within a context. Regarding the time domain, when the running thread is sitting idle, the processor resources cannot be utilized to waiting threads, and, as a result, the processor throughput is severely degraded. For the space domain, there is no guarantee that the scheduled thread is to utilize all the available resources at once. For instance, if a thread is running an integer code, it is quite possible that all the floating-point Arithmetic Logic Units (ALUs) would stay underutilized for a long time.

An SMT processor is a slightly modified version of a superscalar processor. Its main purpose is to avoid resource underutilization at all costs. By upsizing and replicating the resources of a superscalar processor and allowing instructions from multiple threads to enter the processor within a context, an SMT processor solves

resource utilization problems described above. In these processors, when a thread starts sitting idle, there is a high chance that another thread may utilize the resources. Consequently, the processor resources are heavily utilized, and the throughput is greatly improved.

There are, of course, some drawbacks of this mechanism. First of all, an individual thread performance might be worse than the performance obtained from a superscalar processor. The main reason for this anomaly is the fine-grain sharing of datapath resources. As a result, the impact of cache conflicts and resulting cache steals are more severe in SMT processors, and a thread may perform much slower than what one may expect. Secondly, an SMT workload (i.e. multiple threads that run simultaneously) is susceptible to some issues that we never experience in superscalar datapaths. For example, if we do not manage the resource distribution well, a single thread may dominate the others. This may trigger thread starvations and all types of fairness related issues.

To solve these issues, there are implicit and explicit types of resource management policies that are proposed in the literature. The implicit policies are closely related with the fetch stage in which the instructions are accepted to the processor pipeline. They are called *implicit* due to the fact that they try to distribute shared resources among threads, implicitly by regulating the frontend of the pipeline. The oldest SMT fetch policy is known as the round-robin scheduler. In this policy, instructions from threads are introduced into the processor in a round-robin fashion. Although, it is simple to implement in hardware, its performance is usually below average. There are many implicit policies that are proposed in the literature, and we try to address them in Section 2. The alternative mechanisms, the explicit resource management policies, have direct control on which thread gets which portion of a resource; and, therefore, they are more directly involved in resource distribution among threads [1,2,3].

Speculative Instruction Window Weighting (SIWW) mechanism to manage SMT resource usage is recently proposed, emerging as a resource allocation policy which operates both implicitly and explicitly [4]. Basically, SIWW speculatively assigns different weights to instructions residing in the Issue Queue (IQ), and keeps the accounting of cumulative weights for each running thread. The fetch policy is modified so that a thread with minimum amount of cumulative weight is preferred for the instruction fetch whereas threads with cumulative weights exceeding a predetermined threshold are all fetch-gated.

Our proposed method, History-based Predictive Instruction Window Weighting (HPIWW), is based on a similar idea. In this study, we claim that there is a much simpler way to learn the cumulative weights of threads rather than speculating almost everything. We show that SIWW is based on multiple power-hungry speculation mechanisms. While, SIWW circuitry dissipates almost as much power as the Physical Register File (PRF), which is one of the hot spots on contemporary processors, HPIWW circuitry has negligible impact on overall processor power. Moreover, HPIWW performs 3% better than SIWW across all simulated workloads, on the average.

The organization of the rest of this paper is as follows: After laying out the current state of the art in Section 2, we describe the details of our proposed design in

Section 3. Section 4 is focused on the experimental methodology that we carry out, followed by the results and discussion on our tests in Section 5. Finally, Section 6 concludes our study.

2 Related Work

There are both implicit and explicit types of resource management policies that are proposed in the literature. Implicit resource management policies [5,6] try to distribute resources among the threads during the fetch stage without any feedback mechanism about the effect of a resource distribution on processor performance. ICOUNT, STALL and FLUSH are the most commonly known implicit resource management policies.

ICOUNT [5] uses the information on the number of instructions of a thread sitting in the IQ to decide which thread deserves the largest portion from the shared resources. Specifically, the thread which has the fewest instructions in the IQ is to be favored. However, the performance problems due to a thread experiencing high cache miss rates are not addressed well in this policy. Thus, a shared resource can easily be monopolized by a slow thread.

STALL [6] is another fetch technique, which is built on top of the ICOUNT policy. STALL introduces the fetch gating mechanism, which prevents instructions of a chosen thread to enter the processor pipeline. Basically, this technique recognizes the threads with pending cache misses as slow, and applies fetch gating on those threads until they are free from such memory instructions.

STALL is a clever technique for blocking slow threads. However, when a thread is fetch gated there may already be multiple instructions with pending cache misses tying up all resources in the processor. FLUSH [6] is an extension of STALL that tries to overcome this situation. It flushes all instructions of such a thread and makes the resources available to other threads.

All implicit SMT resource management techniques try to distribute resources among threads by altering the SMT fetch policy. As a result, they only focus on one end of the processor datapath, and their hardware cost is almost negligible. However, there is a considerable latency between the selection of a new resource distribution policy and the observation of its actual outcome. Unfortunately, threads can instantly change their behavior, and adaptation of an implicit resource management policy to the instant needs of threads becomes a major issue to be solved. To target all these deficiencies, the explicit resource management policies focus on more direct control of the resource distribution mechanism. They require more complex hardware, but they can micromanage the shared resources at any instant. Dynamically Controlled Resource Allocation (DCRA) [1], learning-based Hill Climbing [2], Adaptive Resource Partitioning Algorithm (ARPA) [3] and Speculative Instruction Window Weighting (SIWW) [4] are the most commonly known explicit resource management policies up-to this date.

Dynamically Controlled Resource Allocation (DCRA) mechanism relies on several runtime statistics for categorizing threads and resources [1]. When a thread has

multiple pending cache misses, DCRA categorizes such a thread as *slow*; otherwise, the thread is categorized as *fast*. Meanwhile, when a resource is not utilized by a thread, DCRA categorizes that thread as *inactive* regarding that resource; otherwise, it is categorized as *active*. At the end, DCRA explicitly decides which thread receives which portion of a resource considering these thread categories. For instance, *slow* threads receive extra resources by stealing them from *fast* threads. The rationale on this approach is as follows: *slow* threads are slow because they do not have enough resources, and, therefore, there is no harm stealing some resources from *fast* threads and giving them to slow ones; the *fast* ones would hardly get any performance hit, anyways.

Learning-based Hill Climbing is another dynamic resource partitioning method that gives more resources to each thread in trial epochs (i.e. time intervals) [2]. After observing the performance of each thread in its trial epoch, the mechanism selects the most successful thread to get more resources at an anchor epoch. Then, the algorithm restarts from the trial stage. Although, DCRA-like mechanisms rely on the information based on hardware counters, such as cache miss rate, instruction occupancy in IQ, and Instructions Per Cycle (IPC), Hill Climbing monitors direct effect of a change in the performance, thus it uses weighted IPC metric for each thread.

Another explicit resource partitioning mechanism is known as ARPA, which tries to give more resources to a thread that uses those resources more efficiently [3]. ARPA keeps account of the usage of each resource by each thread in a given epoch and use this information to calculate the Committed Instructions Per Resource Entry (CIPRE) efficiency metric. If a thread has a high CIPRE value for a specific resource in an epoch, then for the next epoch, it receives more resources.

SIWW mechanism, which is the inspiration for this study, distributes the resources among the threads based on the burden of each thread on the processor pipeline [4]. This is calculated as the cumulative weight of all in-flight instructions for that thread. The weight of each instruction depends on its effect on the processor. For example weight of a memory operation is always higher than the weight of a simple Arithmetic Logic Unit (ALU) operation. If the weight of each instruction is selected to be 1, and if a thread with a cumulative weight greater than Reorder Buffer size is fetch gated, then SIWW behaves exactly like ICOUNT policy.

More recently, policies for allocating multiple threads to cores in multicore SMT processors are also studied [7], yielding that L1 bandwidth consumption is an important parameter affecting performance. In this study, balancing L1 bandwidth demand of different threads is focused, and static and dynamic thread-to-core mapping approaches are proposed. The static method considers standalone L1 bandwidth demand of threads whereas the dynamic method relies on runtime performance counters and an adaptive mapping strategy. The authors claim that the dynamic policy performs better than the static approach, reaching 6% performance improvement over standard Linux OS scheduler.

Turning on and off datapath resources is also one of the approaches studied for saving SMT processor power [8]. Measuring workload needs to dynamically resize resources is found to achieve 12% power reduction. The datapath resources, whose idle partitions are exposed to adaptive resizing, are issue queue, physical register files,

reorder buffer and load/store queue. For increasing or decreasing the amount of resources, resource occupancy is periodically checked and in each period, resource occupancies collected from previous periods are compared to those of the current period.

Finding the group of jobs which increase processor performance when coexecuted is a problem arising in the job symbiosis scheduling studies for SMT processors. Probabilistic modelling proposed for this problem [9], foresees that whether jobs will produce a performance increasing or decreasing effect, using per-thread cycle stacks and avoiding evaluation of the set of coexecuted jobs. This method decreases job turnaround time by 16% and 19%, on the average, in 2-thread and 4-thread SMT processors, respectively, compared to *sample-optimize-symbiosis* approach, and enables a job to progress in accordance with its single-threaded progress suggested by its share.

3 Proposed Design

3.1 SIWW Model

As stated earlier, our proposed design is based-on the ideas of SIWW. Thus, we first implemented the SIWW model in our simulation environment, as explained below. Fig. 1 describes how SIWW updates cumulative weight of each thread (C_w). C_w is initialized to zero, when the processor starts. During the fetch stage, this value is calculated at once. This type of design enables us to simplify the implementation of the fetch stage in the simulation environment. However, we still keep cumulative weight update statistics during the *decode* and the *execute* stages to mimic the actual SIWW policy for accurately calculating the power dissipation of the SIWW.

C : set of contexts w : cumulative SIWW weight $S(i)$: SIWW weight of instruction i I : set of instructions in issue queue
Initialize: $c_w=0$, $\forall c \in C$ 1. $c_w += S(c_r)$, $\forall c \in C$ and $\forall c_r \in I$

Fig. 1. SIWW updateCumulativeWeight(C, I) function

SIWW method utilizes predictors for load confidence, branch confidence and Memory Level Parallelism (MLP). These predictors are realized maintaining confidence estimation tables and assigning instruction weights considering the historical confidence values stored in these tables. Eq. (1) explains update scheme applied on the branch confidence table (bct). Each entry of bct is a 2-bit saturated counter indexed by the hashed value of the Program Counter (hPC) addressing the current branch instruction. A higher value at a bct entry indicates a greater probability for branch decision being correct, increasing the confidence of the corresponding entry.

Since the saturated counter is a 2-bit counter, $bct[hPC]$ is not allowed to be greater than 3. Eq. (1) is implemented in the *writeback* stage of the processor.

$$c.bct[hPC] = \begin{cases} c.bct[hPC] + 1, & \text{if correct pred. } \wedge bct[hPC] < 3 \\ 0, & \text{otherwise if mispred.} \end{cases}, \forall c \in \mathcal{C} \quad (1)$$

Eq. (2) shows how SIWW assigns the weight of branch instructions. If the branch is being predicted correctly, the bct entry is saturated, and the weight of the branch instruction is set to 2; otherwise, it is set to 8.

$$setWeightOfBranch(hPC) = \begin{cases} 2, & c.bct[hPC] = 3 \\ 8, & \text{otherwise} \end{cases}, \forall c \in \mathcal{C} \quad (2)$$

Eq. (3) explains load confidence table (lct) update scheme. lct is composed of 4 bit saturated counters, each of which is accessed as $lct[hPC]$, again hPC being the hashed PC value addressing the current load instruction. When there is an L2 cache miss, the lct entry corresponding to the current instruction is decreased. Similarly, if the memory access does not result in an L2 cache miss, $lct[hPC]$ is increased. Since lct entries are 4 bit saturated counters, the legitimate values for these entries range between 0 and 15. Eq. (3) is implemented in the *memory* stage. Note that, in these equations the L1 cache is not even considered, since a miss to an L1 cache might be served in a few cycles if there is a hit to the L2 cache.

$$c.lct[hPC] = \begin{cases} c.lct[hPC] - 2, & (\text{l2 miss}) \wedge (c.lct[hPC] \geq 2) \\ c.lct[hPC] + 1, & (\text{l2 hit}) \wedge (c.lct[hPC] < 15) \end{cases}, \forall c \in \mathcal{C} \quad (3)$$

Finally, determining weights of load instructions based on the load confidence estimator is shown in Eq. (4). For load instructions whose $lcd[hPC]$ is greater than or equal to 8, its weight is set to 1. Otherwise, MLP predictor contributes to the weight assignment. $MLPweight$ function that determines the weight of such a load instruction is explained in Eq. (5). Both Eq. 4 and 5 are realized in the *decode* stage.

$$setWeightOfLoad[hPC] = \begin{cases} 1, & c.lct[hPC] \geq 8 \\ MLPweight(c.MLP), & \text{otherwise} \end{cases}, \forall c \in \mathcal{C} \quad (4)$$

$$MLPweight(c.MLP) = \begin{cases} 16, & c.MLP > 1 \\ 128, & \text{otherwise} \end{cases}, \forall c \in \mathcal{C} \quad (5)$$

To calculate the MLP degree of a thread, a Miss Pattern Detection or the Long Latency Load (LLL) prediction mechanism, shown in Figure 2, is used. LLL tables are updated considering the history of L2 cache hits and misses. When a cache miss takes place during the *memory* stage, the left LLL table for the current context (c) jumps to the state where hit counting starts from zero, overwritten value being backed up in the right LLL table. Otherwise, hit counting continues on the left LLL table.

```

if(l2 miss){
    c.LLL[hPC].right←c.LLL[hPC].left
    c.LLL[hPC].left←0
}else // if l2 hit
    c.LLL[hPC].left++

```

Fig. 2. Long Latency Load (LLL) prediction algorithm

The MLP history algorithm, whose details are summarized in Figure 3, increments the MLP counter for the current context (c) when an LLL is encountered in the *decode* stage. An LLL is speculated when the number of hits since the latest cache miss and the number of hits between the latest two cache misses are equal to each other.

```

if(c.LLL[hPC].left==c.LLL[hPC].right)
    c.MLP_counter++

```

Fig. 3. The MLP history algorithm

Finally, the MLP update algorithm, which is given in Figure 4, sets the MLP value of a thread to the number of in-flight LLLs. The algorithm is executed during the *commit* stage where the instruction in the head of the Re-Order Buffer (ROB) retires. When an LLL commits, the future value of the MLP for that context decreases.

```

if (c.ROB[head] is an LLL) {
    c.MLP←c.MLP_counter
    c.MLP_counter-
}, ∀c ∈ C

```

Fig. 4. The MLP update algorithm

3.2 HPIWW Model

As described in the previous subsection, SIWW model requires many speculation structures to get a good estimate on the future weight of each instruction sitting in the processor frontend. However, there is an alternative way to this approach, which is known as the history-based prediction, and many processor structures, such as branch predictors, load value predictors and caches, are based on this alternative approach. We design the HPIWW model by assuming that the recent weight history of instructions could be used to predict the weight of instructions currently residing in the IQ. Basically, the weight (i.e. the number of cycles spent in the IQ¹) of a certain number of executed instructions are kept in a queue, which we call History Weight Queue

¹ This somewhat corresponds to the weight information that is being speculated in the SIWW mechanism.

(HWQ), for each thread, and this is assumed to represent the most up-to-date behavior of a thread. Decision of which thread gets more of a resource in the next fetch cycle is determined by the cumulative weight of each thread. As in SIWW, the thread with a minimum cumulative weight is always favored, and when a thread has a cumulative weight greater than the size of the ROB, it is fetch gated.

The HWQ is implemented as a circular queue per thread with the size of the IQ, and when an instruction is scheduled for execution its actual weight is calculated by subtracting its timestamp cycle, which is assigned when that instruction enters the processor, from the current cycle. Then, its actual weight is inserted into the tail of the HWQ. Since the HWQ is a circular queue, the oldest weight value is overwritten by this youngest weight value.

Figure 5 depicts how the running cumulative weight of a thread is calculated and kept accurate. The idea is to hold the sum of weights of N youngest instructions, N representing the number of instructions of a thread in the IQ. When an instruction exits from the IQ, the occupancy of instructions in the IQ for the corresponding thread is reduced by 1, and, in such a case, the weights of the two oldest instructions contributing to the cumulative weight are needed to be subtracted from the current cumulative weight, and the weight of the currently issued instruction should be added. As a result, the two consecutive weight values pointed by the HWQ_{head} (j and k , in the example) subtracted from the cumulative weight, and HWQ_{head} is moved two entries closer to the HWQ_{tail} . Meanwhile, new entry, let's say p (not shown in the Figure), is added overwriting the value of a , and the HWQ_{tail} is moved one entry forward. At the end, the number of instructions in the IQ for that thread is reduced to 3, and the cumulative weight becomes the sum of three items, l , m , and the youngest weight value, p . Here, notice that the number of entries between HWQ_{head} and HWQ_{tail} is synchronized with the number of instructions of the corresponding thread in the IQ. Also note that it is guaranteed that HWQ_{head} can catch but cannot pass HWQ_{tail} pointer.

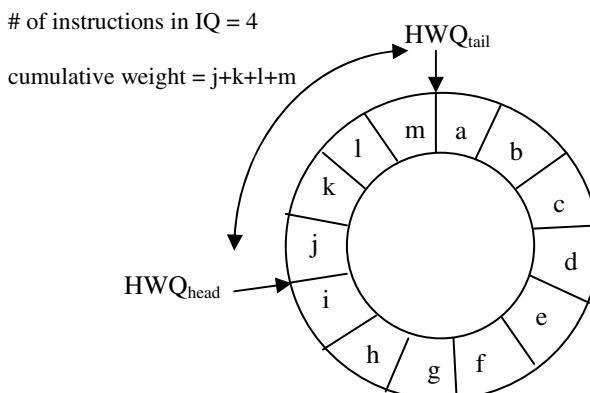


Fig. 5. An example HWQ instance for a thread

Using Figure 5, we also would like to explain the case when an instruction is introduced into the IQ. In such a case, the number of instructions in the IQ for the current thread is increased, but HWQ_{tail} does not move since we cannot calculate the actual weight of an instruction before it is issued. Considering the HWQ instance in Figure 5, we move the HWQ_{head} one entry away from the HWQ_{tail} , accounting the value i in the cumulative weight of the thread.

When the HWQ structures become full, they become fully operational. We suggest using a default fetch policy during these short warm-up periods. However, in those periods, HPIWW can still be utilized, but it may not show its best performance since the cumulative weight information during the warm-up period would not be very accurate.

4 Experimental Methodology

In this section, we provide the details of our experimental setup. To simulate the baseline, SIWW and HPIWW processors, we used M-Sim [10], which is a modified version of the well-known Simple Scalar simulation toolkit. M-Sim can simulate SMT processors and run Alpha binaries. The details of the simulated processors are given in Table 1. SIWW parameters are faithfully implemented to match the parameters of [4], and HPIWW is assumed to utilize per thread IQ-size (32-entry) HWQ structures.

We randomly selected various benchmarks from the SPEC2000 suite, and generated eleven 3-thread workloads, in total. The power calculations are done by the help of M-Sim-integrated Wattch [11] and Cacti [12] tools in 32nm process technology. The cycle-accurate simulations are run for 100M instructions after skipping first 100M instructions.

Table 1. Simulation Parameters

Parameter	Configuration
Machine Width	4-wide fetch, 4-wide issue, 4-wide commit
Window Size	32-entry issue queue, 128-entry reorder buffer, 48-entry load/store queue, 128 Int. phys. reg., 128 FP phys. reg.
L1 I-Cache	32 KB, 2-way set-associative, 32 B line, 1 cycle hit time
L1 D-Cache	32 KB, 4-way set-associative, 32 B line, 1 cycle hit time
L2 Cache (unified)	512 KB, 8-way set-associative, 64 B line, 6 cycles hit time
BTB	512 entry, 4-way set-associative
Memory	100 cycles first chunk, 2 cycles interchunk
TLB	64 entry (I), 128 entry (D), 4-way set-associative, 30 cycles miss latency
Function Units and Latency (total/issue)	4 Int. Add (1/1), 1 Int. Mult (3/1) / Div (20/19), 2 Load/Store (2/1), 4 FP Add (2/1) / FP Cmp (2/1) / FP Cvt (2/1), 1 FP Mult (4/1) / (Sqrt (24/24))

5 Results and Discussion

In this section, we evaluate HPIWW in terms of its power and performance characteristics. Figure 6 shows the normalized power dissipation of both SIWW and HPIWW compared to the well-known hot spot in an SMT processor, the Physical Register File (PRF). The PRF is a heavily ported structure which is read by almost all executing instructions for retrieving the values of their source operands. It is also written, when an instruction has a destination register during the writeback stage. Our studied processor is a 4-way superscalar SMT processor, and it can be read 8 times (4 instructions each with 2 source operands) and be written 4 times (4 instructions writing their destination registers). When Figure 6 is examined, it is clearly seen that SIWW dissipates as much power as the PRF (86% of the PRF to be exact). From this figure we can easily claim that SIWW speculation circuitry becomes a hot spot on the processor. The hot spots are the performance limiters of a processor, since they prevent processors running in higher frequencies. When we examine the HPIWW results, the HWQ structures seem to dissipate negligible amount of power compared to both SIWW circuitry and the PRF.

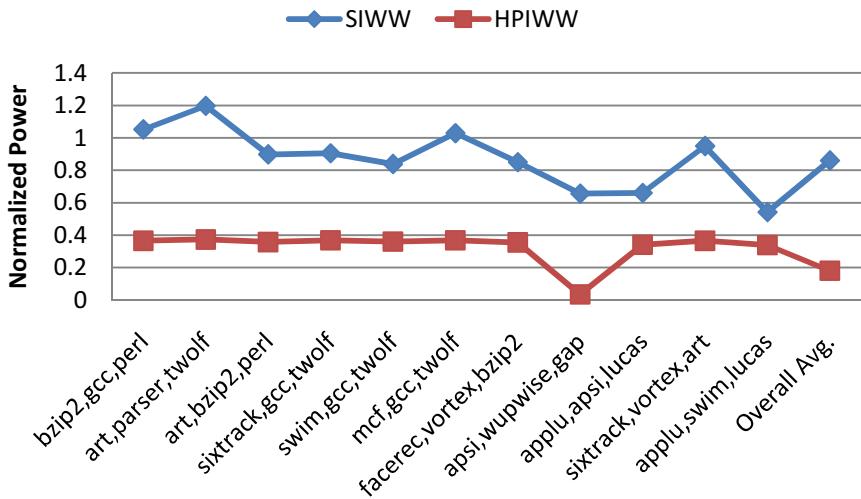


Fig. 6. Power results of SIWW and HPIWW circuitry normalized to the PRF power

When we move to performance results shown in Figure 7, HPIWW is again the clear winner. In all the workloads that we studied, it consistently performs better than SIWW (around 3% better, on the average). SIWW is a successful mechanism by its nature. However, the weight information that is accessed by HPIWW is much more accurate than any speculated weight value of SIWW. As a result, we think that these results verify this basic fact.

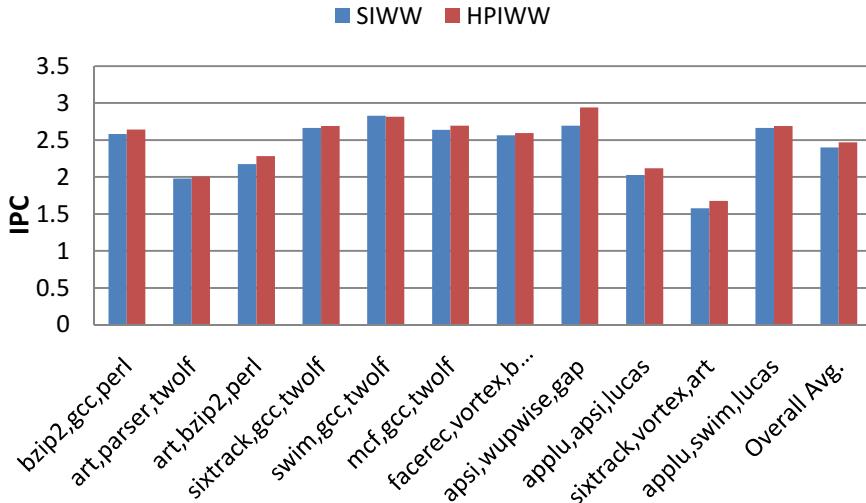


Fig. 7. Performance results of SIWW and HPIWW

6 Conclusion

In recent years, Simultaneous Multi-Threaded processors gain popularity due to their simplicity and noticeable throughput improvements. The most of the research focuses on the fair distribution of shared resources among running threads. When this is somewhat addressed, the throughput is also guaranteed to be improved. However, fairness is an ambiguous term by itself. The well-known ICOUNT fetch policy, for example, tries to balance the number of instructions from each thread within the SMT frontend. By doing so, it aims to be the most fair resource distribution policy to the running threads. Although it looks like it is a fair policy at first sight, the number of instructions of a thread may not be the best indicator of its resource utilization. A single instruction of a thread may be a memory instruction that misses all cache levels, and, as a result, may create resource clog tying up most of the processor resources. On the other hand, another thread may have many instructions that execute almost instantly.

In this paper, we propose a mechanism based on a recently introduced fetch policy named Speculative Instruction Window Weighting (SIWW). SIWW introduces many hardware-based speculation structures that are used to estimate the weight of instructions at the time they are introduced into the processor pipeline. Here, we show that the complex nature of the mechanism dissipates too much power, even though its throughput performance is better than ICOUNT. Then, we introduce the History-based Predictive Instruction Window Weighting (HPIWW) that implements a very similar logic with a negligible amount of additional hardware. Basically, we believe that there is no need for a complex speculation circuitry for gathering the information

that SIWW requires. The much more accurate weight information is already there for each instruction executing in the pipeline at no cost, and HPIWW exploits this fact.

As a result, HPIWW outperforms SIWW in terms of power and throughput metrics. The power savings compared to SIWW is around 2.5 times less, on the average. Since, HPIWW works on much more accurate weight information compared to SIWW, its throughput is also 3% better, on the average across all simulated workloads.

References

1. Cazorla, F.J., Ramirez, A., Valero, M., Fernandez, E.: Dynamically Controlled Resource Allocation in SMT Processors, pp. 171–182. IEEE MICRO (2004)
2. Choi, S., Yeung, D.: Learning-Based SMT Processor Resource Distribution via Hill-Climbing. In: International Symposium on Computer Architecture (ISCA), pp. 239–251. ACM Press, New York (2006)
3. Wang, H., Koren, I., Krishna, C.M.: An Adaptive Resource Partitioning Algorithm for SMT Processors. In: International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 230–239. ACM Press (2008)
4. Vandierendonck, H., Seznec, A.: Managing SMT Resource Usage through Speculative Instruction Window Weighting. Transactions on Architecture and Code Optimization (TACO) 8(3), 12 (2011)
5. 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. ACM SIGARCH Computer Architecture News 24(2), 191–202 (1996)
6. Tullsen, D.M., Jeffery, A.B.: Handling long-latency loads in a simultaneous multithreading processor. In: 34th Annual ACM/IEEE International Symposium on Microarchitecture, pp. 318–327. IEEE Computer Society Press (2001)
7. Feliu, J., Sahuquillo, J., Petit, S., Duato, J.: L1-bandwidth aware thread allocation in multi-core SMT processors. In: Parallel Architectures and Compilation Techniques (PACT), pp. 123–132 (2013)
8. Kucuk, G., Mesta, M.: Energy savings in simultaneous multi-threaded processors through dynamic resizing of datapath resources. Turkish Journal of Electrical Engineering and Computer Sciences 20(1), 125–139 (2012)
9. Eyerman, S., Eeckhout, L.: Probabilistic modeling for job symbiosis scheduling on SMT processors. Transactions on Architecture and Code Optimization 9(2), art. no. 7 (2012)
10. Sharkey, J.J., Ponomarev, D., Ghose, K.: M-SIM: A Flexible, Multithreaded Architectural Simulation Environment. Department of Computer Science, Binghamton University, Technical Report No.CS-TR-05-DP01 (2005)
11. Brooks, D., Tiwari, V., Martonosi, M.: Wattch: A Framework for Architectural-Level Power Analysis Optimizations. In: International Symposium on Computer Architecture (ISCA), pp. 83–94. ACM Press, New York (2000)
12. Wilton, S.J.E., Jouppi, N.P.: CACTI: An Enhanced Cache Access and Cycle Time Model. IEEE Journal of Solid-State Circuits 31(5), 677–688 (1996)

The Brand-New Vector Supercomputer, SX-ACE

Shintaro Momose, Takashi Hagiwara, Yoko Isobe, and Hiroshi Takahara

NEC Corporation, IT Platform Division, Tokyo, Japan

{s-momose@ak,t-hagiwara@ce,y-isobe@pi,
h-takahara@bc}.jp.nec.com

Abstract. Many of the current supercomputers tend to pursue higher peak performance, however, the characteristics of scientific applications are getting diversified, and their sustained performance strongly depends on not only the peak floating point operation performance of the system, but also its memory bandwidth. NEC's goal is to provide superior sustained performance, especially for memory-intensive scientific applications. As the successor to the SX-9, its brand-new SX-ACE vector supercomputer has been developed to achieve this goal. The new vector processor features the world top-class single core performance of 64Gflop/s with the largest memory bandwidth of 64GB/s per core. Four cores, memory controllers, and a network controller are integrated into the SX-ACE processor, enabling the processor performance of 256Gflop/s with its memory bandwidth of 256GB/s. In order to gain a higher sustained performance, the system is equipped with a specialized network interconnecting processors, as well as a sophisticated vectorization compiler and an operating system.

Keywords: Supercomputer, Vector architecture, Memory bandwidth, Bytes/Flop.

1 Introduction

NEC has pursued a higher sustained performance with the SX Series vector supercomputers during the past three decades. After the launch of its pioneering SX-1 and SX-2 models in 1983, NEC's HPC solutions have been represented by the unrelenting evolution of this series toward the recent model SX-ACE, as the successor to the SX-9 [1,2,3,4] with innovative hardware and software technologies. SX-ACE was launched in November 2013, which inherits and improves the SX vector architecture with its design concepts kept unchanged, toward higher sustained performance [3,4] and usability.

A working group of the Ministry of Education, Culture, Sports, Science and Technology of Japan has comprehensively investigated characteristics of various scientific applications with respect to required memory capacity and the required ratio of memory bandwidth to performance (Bytes/Flop, B/F) [5]. Its report points out that the required B/F ratio varies significantly depending on types of applications. Therefore, any single architecture cannot cover all the HPC application areas.

In particular, for memory-intensive applications, such as computational fluid dynamics, weather forecast, and earth science, there is a growing gap between the desired and actual memory capabilities in the case of commodity processor-based systems. Taking into account such a situation, the principle design policy of the SX-ACE processor is twofold. One is to cover memory-intensive application areas through realizing a larger sustained memory bandwidth with reordering memory access instructions and merging redundant memory accesses in addition to its high B/F ratio. Nowadays, the peak performance of the world largest system is expected to increase beyond multi-petaflop, however, the actual memory bandwidth is not commensurate with the peak performance; their imbalance is getting larger for many supercomputers due to the memory wall problem [6,7,8], resulting in lower sustained performance in many practical applications.

The other design concept of the SX-ACE processor is to realize a surpassing single core performance over many of the commodity processor cores rather than increase the number of low-performance, fine-grain cores in the processor. For many modern commodity-based systems, the number of cores increases, while the single core performance remains almost constant [9]. Therefore, an extremely higher parallelization is essential in order to obtain higher sustained performance according to Amdahl's law [10]; however, achieving a parallelization ratio of close to 100 percent is generally quite challenging for many applications. This is due to the required cumbersome parallel programming to eliminate serial portions, as well as complicated communication and synchronization issues among a huge number of nodes or cores. Moreover, in numerical simulations such as those based on the finite difference method, for example, the increased number of processors requires a huge number of computational sub-domains for parallel decomposition, often resulting in increased communications between neighboring sub-domains even with challenging programming techniques. For this reason, the system with a smaller number of cores with a higher single core performance is recommended to allow users to obtain higher sustained performance [11].

From the system point of view, the SX-ACE is equipped with a specialized network for interconnecting processors, a specialized operating system (OS), a sophisticated vectorization compiler, and performance optimization tools in order to gain a higher sustained performance. Most of the modern large-scale supercomputers with a dedicated interconnect network employ an n-dimensional torus network topology. Generally, the torus network can simplify the hardware design with the reduced number of interconnection cables and switches especially for large scale configurations. However it has two drawbacks regarding performance and usability, i.e., lower global communication performance due to a larger network diameter, and complicated programming to mitigate redundant communications that require optimum mapping of scientific applications onto the n-dimensional torus network. To alleviate such issues, the SX-ACE processors are connected with a fat-tree network.

The rest of this paper describes the processor architecture of SX-ACE in Section 2, and its multi-node configuration in Section 3 with the focus on the specialized interconnect network. Section 4 discusses the sustained performance by some benchmark programs, followed by the summary of the paper in Section 5.

2 Processor Architecture of SX-ACE

2.1 Processor Overview

The SX-ACE processor realizes the world top-class core performance and a larger memory bandwidth that inherits the proven SX vector architecture with an improved memory subsystem to gain a higher sustained performance from memory-intensive applications. Figure 1 shows the overview of the processor configuration. It provides a 256GFlop/s (GF) double precision floating-point performance and a 256GB/s memory bandwidth, enabling a B/F ratio of 1.0. While B/F ratios of modern microprocessors are lower than 0.5, such a high memory bandwidth can boost the performance of the memory-intensive applications.

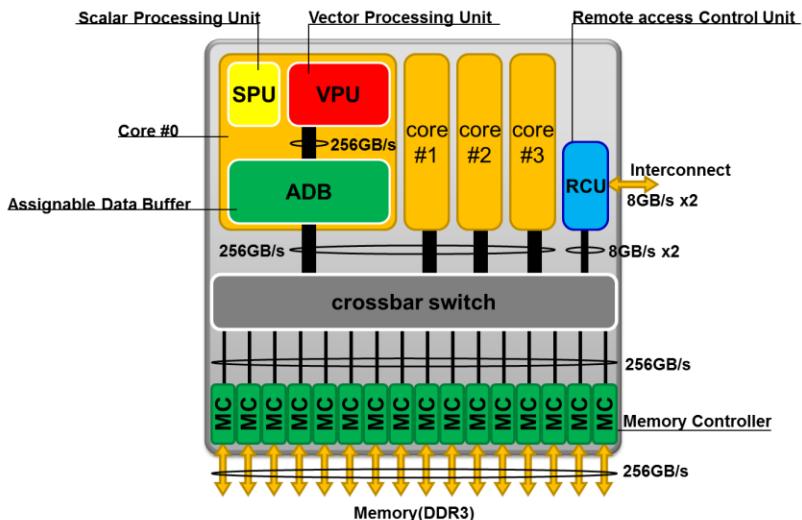


Fig. 1. Processor block diagram

The processor is configured with several major blocks, including four processor cores, 16 memory controllers (MCs) that realize a 256GB/s memory bandwidth, a remote access control unit (RCU), and a crossbar switch. Each core having a performance of 64GF is composed of three major parts; a scalar processing unit (SPU), a vector processing unit (VPU), and an assignable data buffer (ADB). Here, each core can utilize the whole memory bandwidth of the processor of 256GB/s, when the other three cores do not access the memory. Thus, the B/F ratio for a single core ranges from 1.0 up to 4.0 at its maximum, which is quite beneficial to high sustained performances of real applications. MCs have memory interfaces of DDR3 running at 2000MHz. RCU is a remote direct memory access (RDMA) engine that can be directly connected to a proprietary interconnect network at a transfer rate of 8GB/s per direction. These components are connected through the crossbar switch, which is

capable of supplying sufficient data to each component and reducing data transaction conflicts in the data paths.

Figure 2 shows the LSI layout of the SX-ACE processor. Four cores are implemented in the upper and lower areas of the LSI. Each side of the LSI accommodates eight MCs with the DDR3 interface. The crossbar switch is implemented in the central part of the LSI, which is surrounded by four cores and 16 MCs. It reciprocally connects the surrounding components at a high transfer rate of 256GB/s. RCU is implemented in the central upper area, and is connected to the crossbar. This processor is manufactured with the 28 nm process technology. About two billion transistors are integrated into the LSI, with its area of 23.05mm by 24.75mm, running at a 1GHz frequency. Such a relatively low-processor clock cycle contributes to a power reduction in the processor operation.

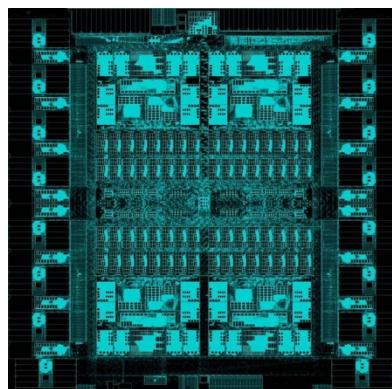


Fig. 2. Processor LSI layout

2.2 Core Architecture

SPU mainly works as a VPU controller and inherits the architecture of 64-bit RISC processors. An independent vector instruction pipeline of SPU is newly employed in order to enhance the capability of issuing vector instructions for VPU. SPU is tightly coupled with VPU to reduce the communication latency between them, and to improve the sustained performance significantly.

VPU is the key component of the SX-ACE vector architecture with its single core performance of 64GF. With the vector architecture of the SX Series including SX-ACE, up to 256 vector elements are processed by one vector instruction. Moreover, VPU has powerful memory access instructions for each set of 256 elements, such as consecutive, stride, and indirect accesses. While the vector architecture is a kind of SIMD, VPU can execute 256 operations in both parallel and sequential ways, which is advantageous in hiding long operation latencies, and differentiates the SX vector architecture from the conventional SIMD architecture.

Figure 3 shows a block diagram of VPU. VPU has three types of vector registers, including vector arithmetic registers (VARs), vector data registers (VDRs), and vector mask registers (VMs), as well as five types of vector operation pipelines. VAR keeps

and supplies operands of operations in the pipeline. VDR holds operands of subsequent operations, supplies operands to VAR, and receives results from pipelines. VM is used for vector mask operations. SX-ACE maintains such a register configuration to ensure the compatibility with legacy object codes. These three types of registers are implemented into 16 vector pipeline processors (VPPs) in an interleaved manner. As VAR and VDR accommodate 256 4B/8B data elements, the total sizes of VARs and VDRs are 16KB and 64KB, respectively. VMs are composed of 16 sets of 256-bit-long registers.

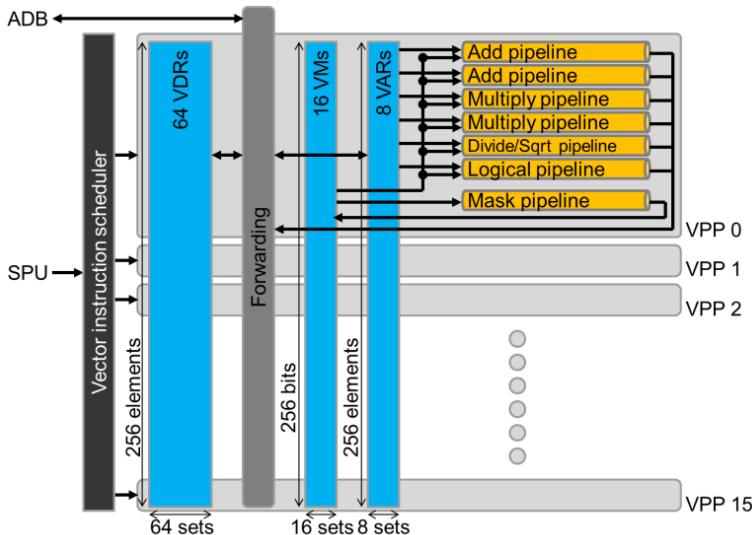


Fig. 3. VPU block diagram

While such large-capacity vector registers contribute to enhanced sustained performance, the increased register size and number of register ports require larger power consumption and die area. VPU employs a multi-banked register in order to realize a larger register size under power/area constraints. Portions of VARs and VDRs in one VPP consist of 8 and 4 sub-portions, respectively, for reduced power/area of registers by exploiting the SRAM with a smaller number of ports. Each sub-portion of VAR and VDR is implemented with 2R2W and 1R1W SRAMs, respectively.

The VPU consists of 16 VPPs, each of which has two add pipelines, two multiply pipelines, one divide/square root pipeline, one logical pipeline, and one mask pipeline as shown in Figure 3. As a result, VPU has 32 sets of add and multiply pipelines, and 16 sets of divide/square root, logical, and mask pipelines. The vector instruction scheduler can control all of 16 VPPs. VPU executes vector instructions by using these 16 VPPs. A vector instruction is capable of handling the vector length of up to 256 that is executed by 16 VPPs in 16 processor clock cycles in both parallel and sequential ways. The peak floating-point operation performance of VPU reaches 64GF by executing 32 multiply-add operations (2-add & 2-multiply per VPP x 16 VPPs) at 1GHz.

For reducing required power and area, VDRs pass data to the pipelines via VAR, and each portion of VARs in a VPP is connected to a corresponding pipeline. Based on the hardware/software architectural co-design, a sophisticated proprietary vector compiler can efficiently handle such complex data transmissions without significant programmers' efforts, enabling higher sustained performance.

Each core has its dedicated 1MB ADB for providing necessary data to VPPs with a larger bandwidth. While the features of ADB are similar to conventional cache, ADB is software controllable for mitigating pollution by non-reusable data and can retain only reusable data, resulting in a higher ADB utilization ratio. The bandwidth between ADB and VPU is 256GB/s, which means 4.0 as its B/F ratio. The LRU (Least Recently Used) replacement policy is employed for the four-way set associative control. Note that the ADB can directly be accessed by VPU with the word-level granularity. Therefore, ADB enables efficient random memory accesses, such as stride and indirect types, resulting in a higher sustained bandwidth for both stride and indirect accesses.

2.3 Memory Controller (MC)

One of the key features of the SX-ACE processor is a large memory bandwidth, which is realized by the unique design of its MC. 16 MCs are implemented in the processor to provide both a larger memory bandwidth of 256GB/s per processor and a larger memory capacity of 64GB per processor. Each MC has a DDR3 interface with a 16GB/s memory bandwidth, and is connected to a dedicated DIMM, which has a 4GB capacity. The data allocation to these 16 DIMMs is interleaved on a 128B block basis. Moreover, MCs have a function of improving a sustained memory bandwidth by reducing memory transactions for store operations, which will be described in the next subsection.

2.4 Memory Subsystem

The major design policy of the SX-ACE processor is to realize efficient memory access with lower energy consumption. There are two functions for this purpose. One is the dual-granularity memory access mode (128B and 64B) for reduced DRAM power by maintaining a high sustained memory bandwidth. The other is the adoption of mechanisms for efficient memory transactions, such as the software-controllable ADB, request compression functions for eliminating redundant memory transactions, and out-of-order memory access functions for enhance sustained memory bandwidth.

To save the power consumption in DRAM accesses, the SX-ACE processor supports two memory access granularities of 64B and 128B. The Micron DDR3 SDRAM system-Power calculator 0.96 [12] indicates the relationships between access granularity and the required DRAM energy. The power needed for activating/pre-charging DRAM strongly depends on access granularity. While a larger access granularity (256 B for example) helps reduce the power for activating DRAM particularly for consecutive memory access, it increases redundant memory transactions due to stride and indirect vector memory accesses. To alleviate such a trade-off, the SX-ACE processor

adopts a 128B access granularity as default, which can reduce the power for activation by 35% compared to the 256B access granularity. To improve sustained memory bandwidth for larger stride and indirect memory accesses, the SX-ACE processor also supports the 64B access granularity.

The SX-ACE processor employs three major techniques in order to avoid redundant memory accesses, i.e., the adoption of ADB and a miss status handling register (MSHR) [13], as well as a request compression function for store operations. The ADB feature is inherited from the SX-9 with the quadrupled 1MB per-core capacity. MSHR and the store request compression function are newly employed. ADB is implemented in each core to reduce the number of memory transactions, enhance the memory bandwidth, and shorten the memory latency between ADB and VPU. The major advantage of ADB over conventional caches is its software controllable data replacement strategy. The user can assign only reusable data to ADB by inserting directives in the source code of the application program to avoid pollution on ADB by non-reusable data. The compiler of SX-ACE also provides a function of automatic data assignment to ADB, which is activated through a bypass flag on each vector memory access instruction. This flag controls whether data are assigned to ADB or not.

In addition, MSHR also reduces the number of memory accesses. MSHR withholds in-flight load requests on ADB misses, and avoids redundant memory requests if the subsequent memory requests that cause ADB misses can be solved by the in-flight load requests. Avoidance of redundant load requests allows the memory bandwidth to be utilized efficiently. MSHR is built into 16 sub-modules of ADB, and 252 redundant load requests can be held in each module. Since MSHR can independently issue 32 in-flight load requests to the memory, the SX-ACE processor can issue up to 512 memory requests in total for the memory.

The load/store request compression functions can also reduce memory transactions by merging several requests to the same cache line into one merged request. The load request compression function is provided by VPU, which merges up to 16 load requests, 8B each, to the same cache line into one 128B load request before the ADB look-up. The store request compression function is provided by a store combining buffer implemented in MCs. The store combining buffer merges up to 16 store requests, 8B each, to the same cache line into one 128B store request, reducing redundant memory transactions.

Two types of out-of-order memory access functions are introduced into the SX-ACE processor through hardware and software controls. The hardware function of the out-of-order memory access checks the memory access dependency among vector load (VLD) and vector store (VST) operations. Each of VLD and VST handles up to 256 elements per instruction. When there is no dependency among instructions, the subsequent instruction can be issued so as to overtake the precedent instruction. Figures 4 (a) and (b) show the mechanism of such hardware-controlled, out-of-order memory accesses. In the case of consecutive accesses, the hardware function checks the addresses of the upper and lower bounds of each instruction. When two instructions are completely independent, the subsequent instruction overtakes the precedent one. In the case of a stride memory access, the function checks the start address and the stride length for each set of 256 elements. A subsequent stride instruction can

overtake a precedent one, when there is no overlapping between them. This feature helps hide memory latency effectively, since subsequent load accesses can overtake thousands of the other precedent memory accesses. This latency hiding mechanism also increases the sustained memory bandwidth.

The software function of the out-of-order memory access enables software-controllable reordering in indirect memory accesses. The indirect memory access instructions such as vector gather (VGT) and vector scatter (VSC) instructions tend to stall other instructions. This is because the hardware-controlled dependency check mechanism for indirect memory accesses, 256-element each, is complicated, and takes many clock cycles. Figure 4 (c) shows an example of the software-controllable out-of-order memory access. A vector overtaking (VO) flag is introduced for VST and VSC instructions. The VO flag allows VST/VSC to be overtaken by other instructions.

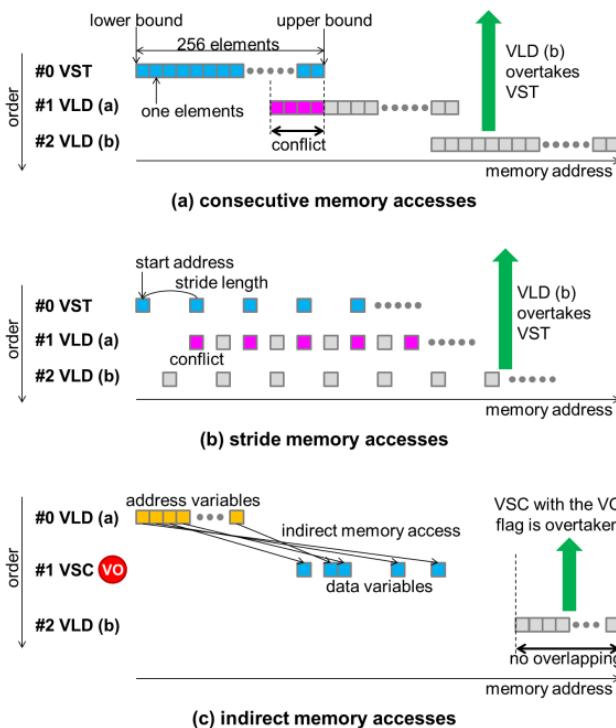


Fig. 4. Out-of-order memory accesses

3 Multi-node System

3.1 Global Network

An SX-ACE single node consists of one SX-ACE processor and 16 DIMMs. Figure 5 shows an example of the SX-ACE multi-node system configured with 512 nodes.

Each node is connected by the specialized interconnect network called global network (GNW) at an 8GB/s throughput per direction, and is composed of the inter node crossbar switch (IXS) and RCU. Ixs is composed of two independent fat-tree topological Ixs planes, which are configured by dedicated router (RTR) switches and cables. In each Ixs plane, 16 nodes are connected to one edge RTR, and all edge RTRs are connected to all spine RTRs. One RTR is composed of 32 ports and a 32 by 32 crossbar switch that controls data packet transfer by referring to each packet header. Communications among nodes are controlled by both RCU and RTRs by using two Ixs planes.

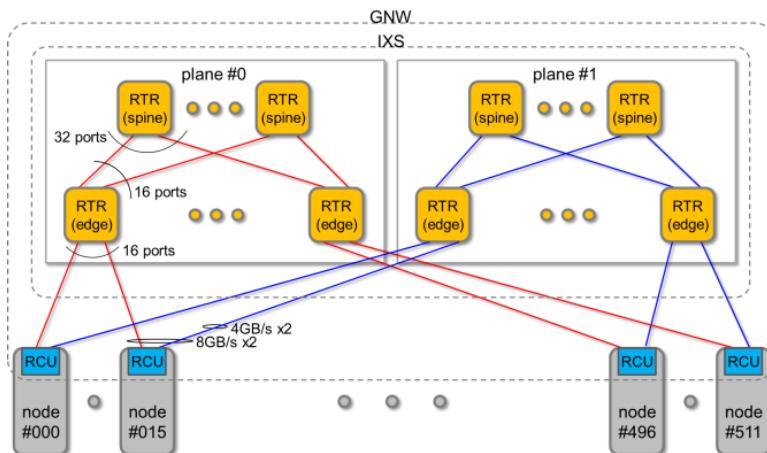


Fig. 5. Multi-node system configuration

RCU is composed of an internode transfer controller, a global address translator, a data transmitter, and a data receiver as shown in Figure 6. The internode transfer controller receives internode access (INA) instructions from cores implemented in the same processor, issues corresponding data transfer instructions to the global address translator, and proceeds internode data transfer through IXS. The global address translator translates destination node addresses from the logical space to the physical one before commencing the transfer by referring to a global node address translation buffer on a source RCU. The data transmitter and the receiver transfer data between the node memory and IXS at 8GB/s throughput per direction with the 2-lane configuration. Thanks to RCU with such an RDMA function, it can work independently from four cores implemented in its processor. The INA instruction is processed by two RCUs in both source and destination nodes by utilizing a global virtual memory address space, which is configured by a logical node number in the system and a virtual memory address in the node.

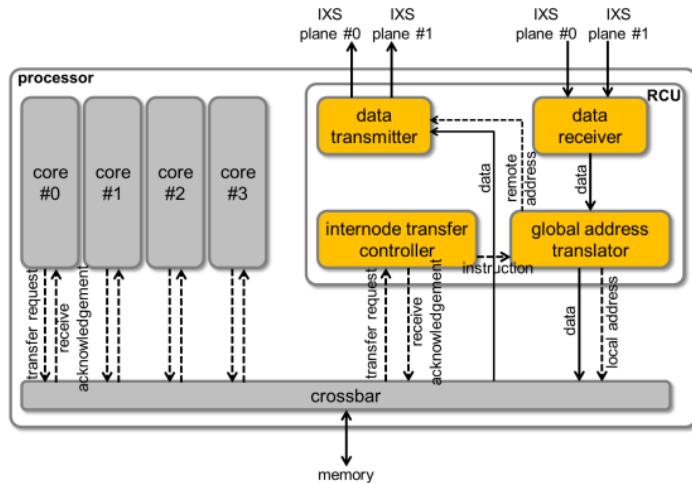


Fig. 6. RCU configuration and data/control signal transferring in a processor

3.2 Internode Communication Functions

GNW provides three types of data transfer functions among nodes, an atomic transfer, an eight-byte transfer, and a block transfer. RCUs process these functions with a small setup overhead for initiating a transfer. The atomic transfer function sends one word of eight bytes in a memory of a source node to that of a destination node as an atomic operation. INA instructions of an asynchronous remote node access are provided for controlling the internode atomic operation. The eight-byte transfer function sends one word of 8B on a register built into a core in a source node to the memory of a destination node without copying the data from the register to the memory in a source node.

The block transfer function transfers block data between a source node and a destination node. The maximum block size handled by one INA instruction is 32MB. There are two types of transfer, a write transfer for sending block data from a source node to a destination node, and a read transfer for sending block data from a destination node to a source node.

In order to accelerate internode global communications, three shared resources are implemented in GNW, i.e., a Global Communication Register (GCR), a Global Barrier synchronous Counter (GBC), and a Global Barrier synchronous Counter Flag (GBCF). GCR is implemented in RTRs and consists of 4K sets of 64-bit registers in a system. GCR is used for synchronization/exclusive controlling among nodes. GBC is also implemented in RTRs and consists of 128 sets of 13-bit registers in a system, and is used for fast internode global barrier synchronization. Each RCU has 128 GBCFs corresponding to 128 sets of GBC registers in the system. The internode global barrier synchronization is executed by utilizing both GBC and GBCF in the following steps.

- Step 1: The number of nodes utilized in parallel execution is set to GBC.
- Step 2: Each node, which has just reached a global barrier synchronization point in the program, decrements GBC.

- Step 3: IXS broadcasts a completion notice of the global barrier synchronization to all RCUs associated with this execution when GBC value attains zero. This indicates that all associated nodes have just reached the barrier synchronization point in the program.
- Step 4: Each RCU receives the notice and enables its own GBCF.
- Step 5: Each node recognizes that the global barrier synchronization has finished by the status of GBCF.

Furthermore, in order to obtain the best possible communication performance, the specialized SUPER-UX operating system and MPI libraries (MPI/SX and MPI2/SX) are also provided.

4 Performance Evaluations

We evaluate the sustained performance and power efficiency of the SX-ACE single node for three fundamental memory-intensive benchmark programs, which are the STREAM benchmark [7], the Himeno benchmark [14,15], and a benchmark program for the Legendre transformation. The STREAM benchmark evaluates the sustained memory bandwidth by disabling data caching functions such as ADB in accessing a data set that has no data locality. The Himeno benchmark is designed to measure the performance in solving the Poisson equation with the Jacobi iterative method, which is highly memory-intensive. This benchmark is used for the evaluation of ADB and MSHR. A practical application kernel program is also prepared to measure the performance of the Legendre transformation that often appears in such an application as numerical weather prediction based on the spectral method, and is used for evaluating the performance of indirect memory accesses. In these three evaluations, the NEC vectorization compiler automatically generates optimal object codes tailored to the SX-ACE processor, thus allowing the data to be allocated to ADB optimally. Furthermore, we evaluate effects of the internode global barrier synchronization function implemented into GNW.

Table 1. Specifications of Evaluated Systems

Name	Processor Frequency (GHz)	Core performance (GF)	ADB capacity per core (MB)	# of cores per processor	Processor performance (GF)	Memory bandwidth per processor (GB/s)	MSHR function	system power consumption per processor (W)
SX-ACE	1.0	64.0	1.00	4	256.0	256	implemented	469
SX-9	3.2	102.4	0.25	1	102.4	256	n/a	1875

Table 1 shows the system configurations that are used for these experiments. We compared the sustained performance of the SX-ACE processor with its predecessor model SX-9. Compared to the SX-9 processor, the SX-ACE processor has a 2.5x higher peak performance and a 4x larger ADB capacity per core, as well as newly implemented MSHR with the approximately one-fourth of the system power consumption per processor.

Figure 7 shows the experimental results of sustained memory bandwidth and sustained power efficiency evaluated by the STREAM TRIAD benchmark in comparison with those of the SX-9. As shown in Figure 7 (a), the SX-ACE processor can provide an approximately 220GB/s sustained memory bandwidth for a varying number of cores, from one to four. It is demonstrated that the SX-ACE processor is capable of fully exploiting its excellent memory bandwidth in contrast to current multi-core processors [16]. While the SX-ACE is designed with the multi-core architecture, even a single core of its processor can utilize the whole memory bandwidth. Although the sustained memory bandwidth of SX-ACE is approximately 10 percent lower than the SX-9 due to the adoption of the DDR3 memory interface as indicated in Figure 7 (a), the DDR3 interface also allows the SX-ACE processor to give approximately 3.5 times higher sustained power efficiency than the SX-9 (Figure 7 (b)). Here note that the obtained sustained memory bandwidths are almost the same regardless of the number of cores used, as shown in Figure 7 (a). These experimental results indicate that the SX-ACE processor can enhance the B/F ratio from 1.0 to 4.0 without relying on ADB by reducing the number of activated cores from four to one. It enables high sustained performance and low power consumption for real applications that access a data set having a size beyond the capacity of ADB with a high B/F ratio.

We also evaluate the sustained performance of SX-ACE by using the Himeno benchmark program as shown in Figure 8. SX-ACE can provide an approximately 1.2 times higher performance in utilizing just a single core of the processor, even though its STREAM performance is lower than the SX-9. The efficient memory subsystem of SX-ACE especially with the enhanced ADB capacity and the newly implemented MSHR function can effectively handle complex memory access patterns appearing in this benchmark program. Furthermore, due to the efficient memory subsystem, SX-ACE realizes a sustained memory bandwidth approximately 3.6 times as large as the SX-9 in spite of the same memory bandwidth of 256GB/s per processor. The major reasons behind this higher performance are 1) the compiler optimally allocating reusable data to ADB, 2) the enhanced ADB capable of accommodating a larger amount of data as compared to the SX-9, and 3) ADB/MSHR for reducing redundant memory transactions. Figure 8 (b) shows the excellent sustained memory bandwidth, as well as higher power consumption efficiencies of 4.8x to 14.6x than the SX-9, for a varying number of cores used within a processor. These results demonstrate that the sustained performance of real memory-intensive applications with some degree of data locality can be boosted by ADB and MSHR in contrast to the STREAM benchmark. In addition, obtained sustained power efficiency ratios are much higher than those for the STREAM benchmark because of effective utilization of ADB and MSHR.

The sustained performance of indirect memory accesses is evaluated by using the Legendre transformation benchmark program as shown in Figure 9. SX-ACE provides 1.9x to 6.3x higher performances than the SX-9, as shown in Figure 9 (a). The efficient memory subsystem with ADB, MSHR, and the out-of-order memory access can accelerate indirect accesses to the memory even with the same memory bandwidth of 256GB/s per processor. ADB allows array data with the locality to be retained, enabling faster accesses to each targeted element of the array through indirect memory accesses. In the event of any data spill from ADB, MSHR can suppress

redundant memory accesses, resulting in a high sustained memory bandwidth even for such sparse data patterns. Moreover, the out-of-order memory access function can prevent unexpected interruption in issuing memory access instructions to ADB or the memory. As shown in Figure 9 (b), compared with the SX-9, 7.4 to 25.3 times higher power efficiencies are obtained. These results clearly show that typical HPC applications with frequent indirect memory accesses can well benefit from SX-ACE with higher sustained performance and lower power consumption.

While some earlier investigations [3,4][11] demonstrate higher sustained performance of the SX-9 for memory-intensive applications than other contemporary processors having a lower a B/F ratio, the results in this study shows well surpassing performance of SX-ACE over the SX-9. It suggests high sustained performance of SX-ACE as compared to other commercially available processors.

We also evaluate the GNW function for the fast internode global barrier synchronization with the increasing number of nodes from two to 512, as shown in Figure 10. We compare execution times for a different number of nodes between a hardware barrier synchronization provided by the GNW function, and a software barrier synchronization without the GNW function. The GNW function gives much shorter execution time than the software barrier synchronization. Note that the execution time of the hardware barrier with 256 nodes are shorter than that with software barrier in the case of two nodes.

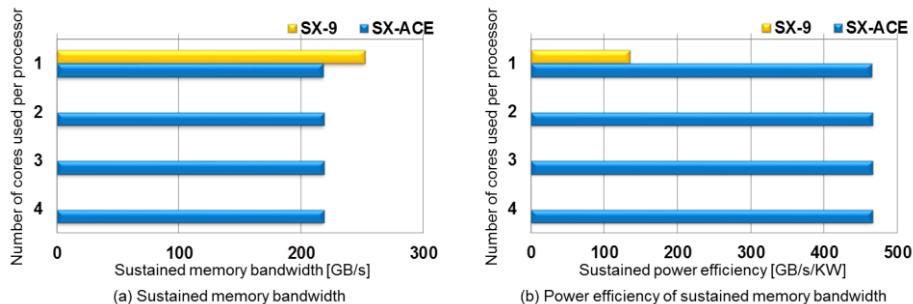


Fig. 7. Performance evaluation results with the STREAM benchmark

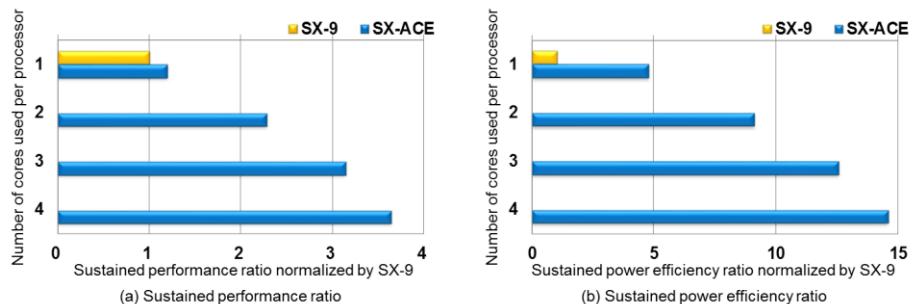


Fig. 8. Performance evaluation results with the Himeno benchmark

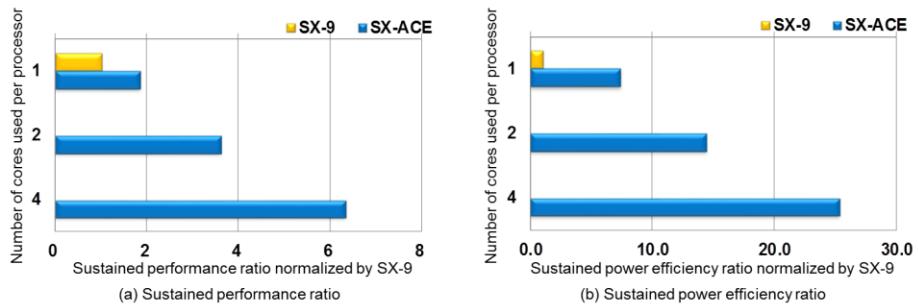


Fig. 9. Performance evaluation results with the Legendre transformation kernel

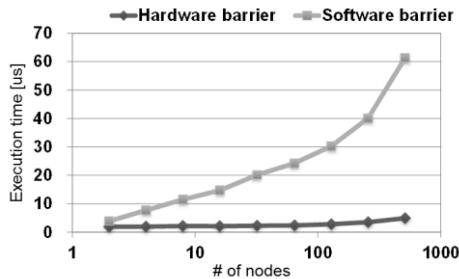


Fig. 10. Performance evaluation result with and without the global barrier synchronization function

5 Conclusions

In this paper, we have presented SX-ACE by focusing on its processor architecture, implementation, and GNW interconnecting nodes. The proven vector processor architecture is inherited from the previous SX series models with two major design concepts of high performance cores and large memory bandwidth. They are keys to enhanced sustained performance for real scientific and engineering applications, such as earth environmental modeling and computational fluid dynamics.

The SX-ACE processor core is characterized by a high single-core performance of 64GF and a large memory bandwidth of 64GB/s to 256GB/s at its maximum. The SX-ACE processor consists of four cores with an aggregate 256GF performance and a 256GB/s memory bandwidth with 16 DDR3 interfaces. Its B/F ratio is 1.0; moreover, ADB can enhance this ratio even up to 4.0. An efficient memory subsystem is introduced for enhancing sustained memory bandwidth by reducing redundant memory transactions. ADB, MSHR, load and store request compression functions, and the out-of-order memory access mechanism can boost the sustained memory bandwidth. Each node with one processor is connected by GNW with an 8GB/s throughput per direction as a multi-node system. Three shared resources in GNW, such as GCR, GBC, and GBCF, provide fast internode global barrier synchronization.

In the fundamental evaluation by using three memory-intensive benchmark programs, we found that 3.5x to 25.3x higher power efficiency than the SX-9 can be obtained. In the GNW functional evaluation of the internode global barrier synchronization, 16x shorter execution time than the software barrier execution on GNW is obtained.

Acknowledgments. We would like to express our cordial gratitude to Cyber Science Center at Tohoku University for the intensive performance evaluation of the SX vector supercomputers as part of the research project with NEC Corporation. In particular, we would like to acknowledge special efforts by Professor Hiroaki Kobayashi, Associate Professor Ryusuke Egawa, and Assistant Professor Kazuhiko Komatsu.

References

1. Satoshi, N., Satoru, T., Norihito, N., Takayuki, W., Akihiro, S.: Hardware Technology of the SX-9 (1) -Main System-. *NEC Technical Journal* 3(4), 15–18 (2008)
2. Takahara, H.: NEC SX Series Vector Supercomputer. In: *Encyclopedia of Parallel Computing*, vol. 4, pp. 1268–1277. Springer (2011)
3. Soga, T., Musa, A., Shimomura, Y., Itakura, K., Okabe, K., Egawa, R., Takizawa, H., Kobayashi, H.: Performance Evaluation on NEC SX-9 using Real Science and Engineering Applications. In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pp. 1–12 (2009)
4. Zeiser, T., Hager, G., Wellein, G.: The world’s fastest CPU and SMP node: Some performance results from the NEC SX-9. In: *Proceedings of IEEE International Symposium on Parallel & Distributed Processing (IPDPS 2009)*, pp. 1–8 (2009)
5. Working group of the Ministry of Education, Culture, Sports, Science and Technology of Japan, White Paper for Strategic Direction/Development of HPC in Japan (2012)
6. Prabhakar, R., Vazhkudai, S.S., Kim, Y., Butt, A.R., Li, M., Kandemir, M.: Provisioning a Multi-tiered Data Staging Area for Extreme-Scale Machines. In: *Proceedings of 31st International Conference on Distributed Computing Systems, ICDCS* (2011)
7. McCalpin, J.: Memory bandwidth and machine balance in current high performance computers. In: *IEEE Computer Society Technology committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25 (1995)
8. Kobayashi, H.: Implication of Memory Performance in Vector-Parallel and Scalar-Parallel HEC Systems. In: *High Performance Computing on Vector Systems 2006*, pp. 21–50. Springer, Heidelberg (2007)
9. Kogge, P.M., Dysart, T.J.: Using TOP500 to Trace and Project Technology and Architecture Trends. In: *Proceedings of SC* (2011)
10. Hill, M.D., Marty, M.R.: Amdahl’s Law in the Multicore Era. *IEEE Computer* 41(7), 33–38 (2008)
11. Takahashi, K., Goto, K., Fuchigami, H., Azami, A., Kataumi, K.: World-highest Resolution Global Atmospheric Model and Its Performance on the Earth Simulator. In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 1–12 (2011)
12. Micron System Power Calculators,
<http://www.micron.com/products/support/power-calc>

13. Musa, A., Sato, Y., Soga, T., Egawa, R., Takizawa, H., Okabe, K., Kobayashi, H.: Effect of MSHR and Prefetch Mechanisms on an On-Chip Cache of the Vector Architecture. In: Proceedings of International Symposium on Parallel and Distributed Processing with Applications, pp. 335–342 (2008)
14. The Himeno benchmark, <http://acc.c.riken.jp/2444.htm>
15. Sato, Y., Inoguchi, Y., Luk, W., Nakamura, T.: Evaluating Reconfigurable Dataflow Computing Using the Himeno Benchmark. In: Proceedings of International Conference on Reconfigurable Computing and FPGAs (ReConFig), pp. 1–7 (2012)
16. Kerbyson, D.J., Barker, K.J., Vishnu, A., Hoisie, A.: Comparing the Performance of Blue Gene/Q with Leading Cray XE6 and InfiniBand Systems. In: Proceedings of 2012 IEEE 18th International Conference on Parallel and Distributed System, pp. 556–563 (2012)

Impact of Future Trends on Exascale Grid and Cloud Computing

T.H. Szymanski

McMaster University, Hamilton ON L8S 4K1, Canada

teds@mcmaster.ca

<http://www.ece.mcmaster.ca/faculty/teds>

Abstract. This paper explores the impact of future trends on Exascale Grid/Cloud Computing systems and data-centers, including: (i) next-generation multi-core processors using 14 nm CMOS, (ii) next-generation photonic Integrated Circuit (IC) technologies, and (iii) a next-generation *Enhanced-Internet* 'lean' router. The new low-power processors offer ≈ 100 cores and a large embedded memory (eRAM) on one CMOS IC. Photonic ICs can potentially lower the size and energy requirements of systems significantly. The lean router supports deterministic TDM-based virtual circuit-switching in a packet-switched IP network, which lowers router buffer sizes and queueing latencies by a factor of 1,000. Our analysis indicates that an entire 'lean' router (optical packet switch) can be fabricated on one photonic IC. Exascale roadmaps have called for: (i) energy reductions by factors of 100 by 2020, and (ii) predictive designs of ≈ 200 PetaFlop/sec systems which consume 15 MW by 2015. Using 2015 technology, we present high-level predictive designs of ≈ 100 PF/sec Grid and Cloud systems which use ≈ 13.1 MW for computation, and ≈ 0.5 and 1.5 MW for communications respectively.

Keywords: energy efficiency, Internet, data-centers, exascale, cloud computing, photonics, circuit-switching, QoS, ultra-low latency, scheduling.

1 Introduction

This paper explores the impact of new technologies on Exascale Grid/Cloud Computing systems and data-centers. Governments in the USA, European-Union, China, India and Asia are pursuing Exascale computing initiatives, aiming to achieve Exascale performance by 2020. In this paper, we explore the impact of future technologies being developed by companies such as Cisco, Intel and IBM, and which will be commercially available soon. Using these new technologies combined with a new *Enhanced-Internet* router technology, we present high-level '*Predictive Designs*' for 100 PetaFlop/sec (PF/sec) HPC Grid and HPC Cloud computing systems, using about 15 MW of power using 2015 technologies, focussing on the global networking.

Several roadmaps have outlined the technical challenges to achieving Exascale computing, including the *International Exascale Software Project* (IESP)

Roadmap [1], the US Dept. of Energy (DOE) reports on Exascale and Cloud computing [2–4], and several others [5, 6]. Common challenges include: (i) scalability to millions of CPU cores, (ii) development of a higher-performance energy-efficient communications network, and (iii) development of energy-efficient computational units (CPUs and GPUs). According to the IESP roadmap, power limitations often limit the bandwidth capacity of the global communications network, which in turn results in lower CPU utilizations, lower data-center utilizations and lower energy efficiency ([1], pg 53): *"Power has become the leading design constraint for future HPC systems designs. In thermally-limited systems, power also forces design compromises...such as reduced global systems bandwidth. The design compromises...will reduce system bandwidth and...greatly limit the scope and application of such systems"*. In this paper, we explore new technologies to address the global bandwidth problem.

Several studies estimate the energy use of existing routers at about 10 nJ/bit. The Cisco CRS-1 router uses about 17 nJ/bit [14], while the CRS-3 router uses about 7 nJ/bit [15]. Scaling existing networking technologies to the range of 10 Petabits per second (Pbps) needed for Exascale computing will require powers of ≈ 100 MW, which is unfeasible. As of Nov. 2013, the current Top10 HPC machines including the Cray Titan and the Tianhe-2 (www.top500.org), have energy-efficiencies in the range of 0.5 W per GigaFlop/sec (GF/sec). Scaling existing architectures and technologies to 1 Exaflop/sec will require powers of ≈ 0.5 GW, which is also unfeasible. These roadmaps collectively call for reductions in the energy-requirements by factors of about 100 by 2020, or about 2 orders of magnitude.

The IESP roadmap also calls for '*Predictive Designs*' of pre-Exascale systems to be delivered by 2015. Specifically, the US DOE Exascale report calls for the design of a 200 PF/sec system, with about 20 GB/sec of interconnect bandwidth per node, which consumes 15 MW by 2015 [3]. We present a high-level predictive design of 2 versions of a 100 PF/sec system, the Grid and Cloud versions. Both systems use ≈ 13.1 MW for computations. Using the DOE target of 20 GB/sec of interconnect bandwidth per node, the Grid version uses ≈ 0.5 MW for networking within a data-center, while the Cloud version uses ≈ 1.5 MW for networking across the Internet. The energy used for global networking is surprisingly low, and indicates that the Exascale targets could be met with future technologies, and that HPC cloud computing may be attractive. In the Grid version, the energy used for global communications is reduced to about 10-30 pJ/bit. In the Cloud version, the energy used for global communications in routers is reduced by a factor of 100, to about 125 pJ/bit. Using the newest CMOS processors, the energy used for computation is also reduced by a factor of 10, from about 0.5 W/GF/sec to about 66 mW/GF/sec. Our predictive designs are reasonably consistent with the IESP and DOE design targets specified for 2015, and these technologies will continue to scale well for several more years. We are unaware of any other high-level predictive pre-Exascale system designs for 2015, and hence we have no other benchmarks for comparison. We also note that '*predictive designs*' are like weather predictions; They represent reasonable

predictions of future performance based on the best available information, but ultimately they are predictions.

To achieve these performance figures, our predictive system design explores: (i) a next-generation of low-power multi-core processors using 14 nm CMOS based on the Intel Knight's Landing chip [18]; (ii) next-generation photonic IC technologies, and (iii) a next-generation *Enhanced-Internet* 'lean' router which supports deterministic TDM-based virtual circuit-switching on a packet-switched IP network. The use of circuit-switched connections in the lean router will lower router buffer sizes and queueing latencies by factors of 1,000 [28–30], and enables the use of integrated photonics. The new low-power CMOS multi-core processors include ≈ 70 multiple-issue cores and a large embedded memory (eRAM) on one integrated-circuit, to ease memory bandwidth constraints.

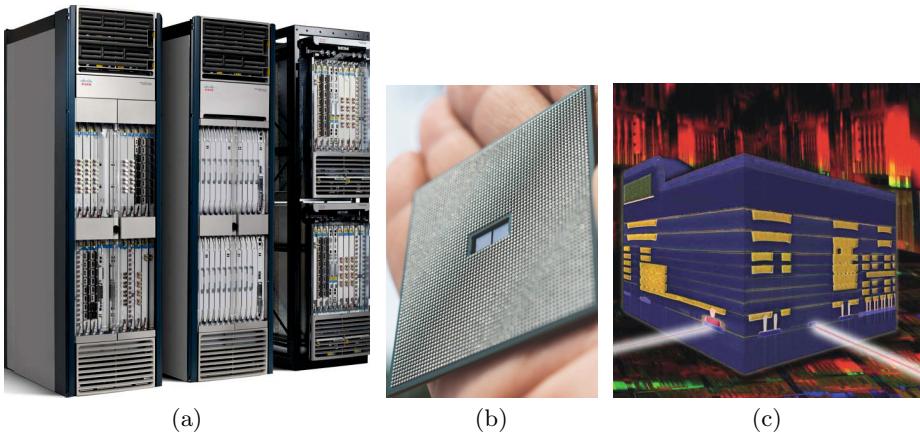


Fig. 1. (a) CISCO CRS-3 router, (b) Compass-EOS CMOS/VCSEL chip, (c) IBM Silicon-Nanophotonics chip

A Cisco CRS-3 router comprising several racks of electronics is shown in Fig. 1a. A CMOS-VCSEL device which implements a large router on one photonic IC is shown in Fig. 1b. This chip was developed by a startup company called Compass-EOS, in which Cisco has invested several \$Million (www.compass-eos.com). A *Silicon-Nanophotonics* IC developed by IBM is shown in Fig. 1c [26] (<http://researcher.ibm.com>). In principle, photonic ICs can lower energy-requirements of communications equipment significantly. However, one major challenge of photonic IC technology is how to incorporate all the complex functionality and buffering associated with a real Best-Effort IP router. For example, using the traditional *Bandwidth-Delay-Product* buffer-sizing rule, routers buffer about 250 millisec of data on each incoming link to mitigate the effects of transient congestion. A conventional IP router with a capacity of 20 Tbps will require about 5 Terabits of high-speed buffering, which is clearly impossible to build on one IC within the next decade. Even Intel's most aggressive CMOS

processors such as the Knight's Landing chip can only hold 16 Gigabytes of slow Dynamic memory. We consider an *Enhanced-Internet* 'lean' router which establishes completely programmable TDM-based virtual circuit-switching connections on a traditional packet-switched IP network [28–30]. The use of virtual circuit-switching can reduce router buffer sizes and queueing delays by factors of 1,000, and result in 95-100% link utilizations, ultra-low latencies, and deterministic and essentially-perfect QoS guarantees. These reduction in buffer sizes also enable an entire 'lean' router to be fabricated one on photonic IC, which unlocks the capabilities of integrated photonics.

Section 2 reviews the challenges of Cloud Computing over the Best-Effort Internet. Section 3 reviews next-generation CMOS processors. Section 4 reviews the next-generation photonic IC technologies. Section 5 presents the single-chip OEO lean router (optical packet switch). Sections 6 and 7 explore 100 PF/sec HPC Grid and Cloud Computing systems. Section 8 concludes the paper.

2 The Challenges of Cloud Computing over the Best-Effort Internet

A primary advantage of Cloud Computing is reduced cost. A cloud computing system can utilize existing Internet data-centers, thereby removing the capital costs of building a dedicated HPD grid computer and its large power generation plant. Internet data-centers are typically under-utilized with typically $\leq 50\%$ utilization [12, 13]. Under-utilized data-centers consume megawatts of power and accomplishing no useful work. A cloud computing system can utilize the idle processing capacity of existing data-centers, thereby improving data-center utilizations and energy efficiencies, while accomplishing useful work at the same time. A Big Data or Exascale Cloud Computing system linking 100 data-centers over the Internet can be interconnect potentially millions of processors, considerably larger than the Cray Titan supercomputer. A cloud computing system is also available on a pay-for-use basis, with potentially lower front-end capital costs of a traditional HPC grid machine like the Cray Titan.

In 2013, researchers at USC and Cycle Computing interconnected 156,000 cores on 5 continents, utilizing about \$68 Million of equipment, to perform 2.3 million CPU hours of computation in 18 hours, at a cost under \$2,000 per hour (www.cyclecomputing.com). However, the application placed minimal communications demands on the Internet. According to the US DOE [4, 7], Cloud Computing is currently not cost effective, and is suitable only for scientific applications with minimal communications and IO requirements. We note that the US DOE study considered only existing computing and Internet technologies, rather than the future technologies considered in this paper.

Traditional HPC machines use tightly-coupled low-latency communication networks such as 10GB Ethernet or InfiniBand which do not drop packets, with average delays in the microsecond range. Today's *Best-Effort* (BE) Internet network is a loosely-coupled network which intentionally drops packets when congestion is detected [8], and it offers poor bisection bandwidth, latency, and

energy-efficiency. Current routers typically use a '*Bandwidth-Delay-Product*' buffer sizing rule, where links in a core IP router typically have buffers for about 250 millisecond of data [8, 9]. These large buffers keep routers busy when arriving traffic is delayed due to transient network congestion. At 1 Tbps, each link in a router requires about 250 Gbits of very-fast buffering, corresponding to about 20.83 million maximum-size IP packets with ≈ 1500 bytes each. These large buffers increase the size, cost and power requirements of existing routers [9]. Given such large buffers, the BE Internet relies upon the significant over-provisioning of bandwidth to keep the average buffer utilization low and to achieve modest end-to-end delays of about 100-200 msec.

The poor performance and excessive delays of the BE-Internet also lowers the utilizations of Internet data-centers for time-critical tasks like Cloud Computing, which further lowers data-center energy efficiency. As a result, governments world-wide are exploring '*Future Internet*' architectures, and they are open to both 'evolutionary' or 'revolutionary' changes to the Internet architecture. Recently, the *Greentouch Consortium* was formed by the telecommunications industry, with a goal of achieving significant reductions in the energy per bit of the Internet [17] (www.greentouch.org). Hence, there is significant interest in technologies to enable energy-efficient cloud communications.

3 Next Generation Processor Technology

In this section, we review the next generation of processor technology. As of 2012, the largest '*High Performance Computing*' (HPC) machine was the Cray Titan supercomputer located in the USA. The Cray Titan has nearly 20K AMD Opteron processor chips and the same number of Nvidia Tesla K20 Graphics Processing Units (GPUs) as co-processors. Each GPU chip has about 7.1 billion transistors using a 28 nanometer CMOS process, provides about 1.3 TFlop/sec (double-precision) and requires about 235 Watts. The Cray Titan has a peak theoretical performance of 27 PF/sec, a peak demonstrated performance of 17 PF/sec, it uses about 8.2 MegaWatts of power, and has a cost of $\approx \$100$ Million US. The GPUs use about 50% of the Cray Titan system power, and the Titan operates at about 63% of peak performance.

Using existing technology and assuming linear scaling, an Exascale HPC system would require about 50 Cray Titan machines, at a cost approaching \$5 Billion US. The system would consume about 20K square meters of floor-space, and require a power plant with about 0.4 GW of power. The US DOE Exascale Initiative Steering Committee has identified a pre-Exascale target system for 2015 capable of about 200 PF/sec of peak performance [3]. One configuration could use about 64,000 nodes with a performance of about 3 TF/sec each, with a node interconnect bandwidth of 20 GB/sec, with 5 PetaBytes of system memory, using 15 MW of power. The goal is 1 day of successful operation between interruptions in 2015, i.e., the *Mean Time To Interruption* (MTTI) ≥ 1 day.

Intel developed the Knight's Landing chip in its push to achieve Exascale computing by 2020, and it is expected to appear in products in 2015 [18]. Advance reports indicate that it has about 5 billion transistors, 72 cores each with

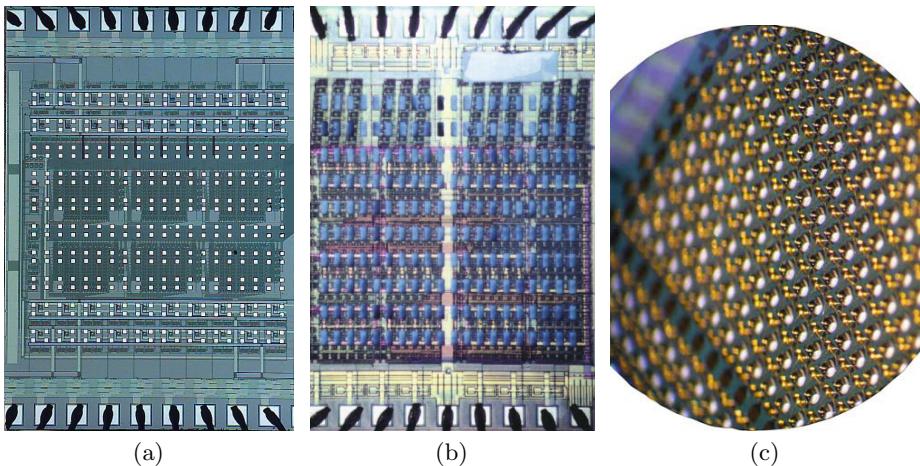


Fig. 2. (a) CMOS substrate before bonding, (b) CMOS substrate after bonding of optical IO, (c) Commercially-available VCSEL arrays (www.tsb-optick.de)

a multi-issue vector unit, and 16 Gbytes of embedded DRAM (eDRAM) on a single 14 nanometer CMOS substrate. It has a peak performance of about 3 TF/sec, rivalling the new Nvidia GPUs, and consumes about 200 Watts.

A next-generation data-center built with 4,096 Knight's Landing chips will offer a peak theoretical performance of 12.3 PF/sec. Using the same parameters as the Cray Titan (i.e., achievable performance $\approx 63\%$ of peak, CPUs using 50% of total power, with the remainder for networking, memory, disks and cooling), the data-center should achieve a performance of ≈ 7.75 PF/sec and require ≈ 1.64 MW for computations. A Cloud system linking 8 small data-centers with 4K processors each will have a peak performance of ≈ 96 PF/sec, and an achievable performance of 62 PF/sec (if the cloud communications can be supported).

By Moore's Law, the performance of computing systems will improve over the next several years. The up-front capital costs and area requirements of an Exascale system will decrease accordingly. However, power requirements are expected to decrease slowly and energy-efficient communications will remain a key design goal.

4 Next-Generation Photonic Integrated Circuits

A *Photonic Integrated Circuit* combines the logical processing of CMOS technologies, with the high-bandwidth IO of lasers and photodetector, on a single integrated package. There are several feasible photonic IC technologies which will begin appearing in commercial systems soon, the (i) CMOS/VCSEL technology, (ii) the Intel *Silicon Photonics* technology [25], and (iii) the IBM *Silicon Nanophotonics* technology [26].

In the CMOS/VCSEL technology, arrays of VCSELs (Vertical Cavity Surface Emitting Lasers) and PDs (Photodetectors) are fabricated on an Indium-Phosphide or Gallium-Arsenide optical substrate, which is flip-chip bonded onto a CMOS substrate. The CMOS substrate contains bonding pads on the top layer of metal, to bond to the optical substrate. Fig. 2a and 2b show a CMOS substrate before and after the bonding with optical IO [21]. Fig. 2c show a commercially-available VCSEL array. The CMOS substrate contains the VCSEL driver circuits and PD receiver circuits, along with a significant amount of CMOS logical processing for the application-specific functionality. The optical transmissions typically emit vertically from the optical substrate and can be captured using for example a medical image guide, which typically has hundreds or thousands of parallel multi-mode fibers arranged in a tube-like structure a few millimeters in diameter. Similarly, the optical streams to be received are typically imaged vertically onto the optical substrate.

The CMOS/VCSEL technology typically uses optical wavelengths in the 850 or 980 nanometer range, it typically uses *Dense Space Division Multiplexing* (DSDM) by transporting thousands of parallel optical transmissions, and it typically uses multi-mode fiber technologies, such as parallel fiber ribbons and medical image guides. It is suitable for short distance communications within a data-center or large router (i.e., ≤ 1 kilometer), i.e., for chip-to-chip, board-to-board and rack-to-rack communications. Using 2014 technology, it is easily feasible to integrate over 4K (64x64) VCSEL and PDs onto an optical substrate. Optical rates of 20 GHz per VCSEL are achievable, providing an potential optical IO bandwidth of 80 Terabits/sec per chip. By 2020, much higher optical bandwidths will be achievable. The technology is well-developed, the CMOS processing is standard and inexpensive, and there are several commercial vendors of VCSEL and PD arrays. Cisco is funding a startup company called Compass-EOS to develop traditional IP routers using this technology.

In the *Silicon Photonics* (SP) technology developed by Intel [25], and the *Silicon Nanophotonics* technology developed by IBM [26], many optical devices and waveguides are integrated into a silicon-on-insulator (SOI) substrate, which contains CMOS logic. The technology allows for integrated planar optical waveguides, switches and integrated optical multiplexers/demultiplexers, allowing for the integration of a complete *Dense Wavelength Division Multiplexing* (DWDM) optical system onto the SOI substrate. Light from laser diodes can be captured into integrated waveguides, which are routed through the SOI substrate, in between the layers of metal used by the CMOS process, as shown in Fig. 2c.

As of 2013, energy-efficient laser diode/photodetector pairs can achieve data-rates of 20 Gbps, using about 7.3 mW of power per Gbps [24]. Using these transceivers, a photonic IC using any of the above technologies with a 1K array (32x32) of VCSEL laser diodes operating at 20 Gbps will have an aggregate IO bandwidth of 20 Tbps, and consume about 150 Watts. The net energy requirement is 7.5 picoJoules/bit (pJ/bit), about 3 orders of magnitude smaller than recently published figures for routers of about 10 nJ/bit. The power requirement will continue to decrease as the CMOS technology feature size decreases with

Moore's Law, at least over the next several years. The reductions in power will increase the optical IO bandwidth realizable on a single photonic IC, for the next several years. The ability to scale CMOS much smaller than 5 nanometers is unclear, so there may be a fundamental upper limit to the optical IO bandwidth achievable per photonic IC, several years into the future.

5 The Lean Enhanced-Internet Router

An *Enhanced-Internet* router which supports deterministic TDM-based virtual circuit-switched (CS) connections, on a conventional packet-switched (PS) IP router, has been presented in [28–30]. The '*Enhanced-Internet*' is essentially a converged circuit-and-packet switched network, combining the advantages of both circuit-switching and packet-switching. The *Enhanced-Internet* router can support 2 traffic classes, a new high-priority *Smooth* class for smooth virtual circuit-switched traffic flows, and the usual *Best-Effort* (BE) class for traditional bursty packet-switched flows. The use of TDM-based virtual circuit-switched connections can lower the router buffer sizes and queueing delays by a factor of 1,000, and achieve *Deterministic and Essentially-Perfect* bandwidth, delay, jitter and QoS guarantees, for all *admissible* traffic demands within the *Capacity Region* of the network [28]. IP links can operate at 95%–100% loads, and still achieve ultra-low latencies and deterministic QoS guarantees. (The lean router can also be operated as a layer 2.5 optical packet switch, which can bypass the Best-Effort IP routers at layer 3.)

An *Enhanced-Internet* lean router is shown in Fig. 3. Each router has 2 classes of *Virtual Output Queues* (VOQs), the *Smooth* VOQs, and the BE VOQs. Each router filters the incoming packets and forwards them to the appropriate VOQs. Packets belonging to bursty TCP/IP traffic flows are forwarded to the BE-VOQs. Packets belonging to smooth traffic flows are forwarded to the Smooth-VOQs. The routing and scheduling of Best-Effort packets through the

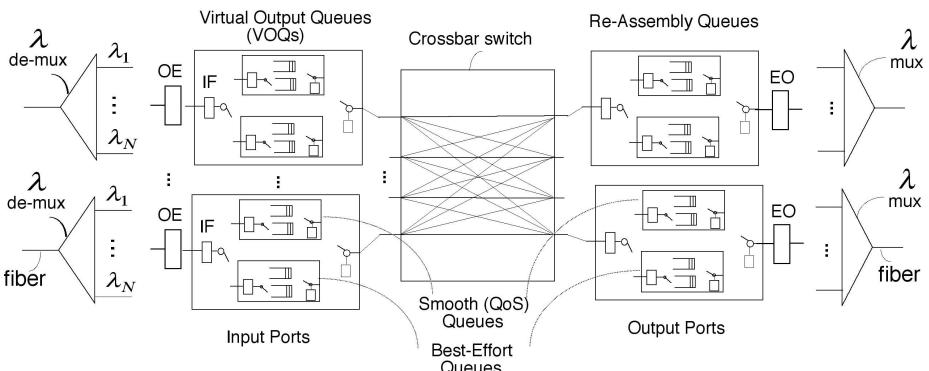


Fig. 3. Enhanced-Internet 'lean' router

router is accomplished with the existing Best-Effort scheduling and routing algorithms (i.e., OSPF), i.e., nothing has changed for bursty Best-Effort traffic. The scheduling of packets belonging to *Smooth* connections through the router is accomplished using deterministic low-jitter TDM-based schedules, which can be precomputed by each router.

Note that the problem of scheduling traffic flows to achieve 100% throughput in an router is difficult, i.e., see [19]. The problem of scheduling traffic flows to achieve minimum jitter in one router is NP-Hard, i.e., see [20]. The problem of scheduling traffic flows to achieve deterministic and essentially-perfect virtual circuit-switched connections along with 100% throughput and low-jitter is an equally hard problem, as it combines both problems. The *Enhanced-Internet* uses a fast polynomial-time low-jitter QoS-aware router scheduling algorithm, to approximate the NP-Hard problem of finding minimum-jitter schedules with 100% throughput. These schedules support completely programmable deterministic TDM-based virtual circuit-switching, on a packet-switched technology. A traffic demand matrix which specifies the smooth traffic demands between the IO ports of each router is specified. The matrix is mathematically decomposed to yield a low-jitter deterministic TDM-based schedule called a Queue-Schedule, which specifies the connections (matchings) to be made by the router for each time-slot in a scheduling frame. The schedules can be computed in microseconds, they support 100% throughput, and they are re-used until the traffic demand matrices changes, i.e., when new connections are added or removed from the routers. Matchings which are not used to transport smooth traffic can be used to transport best-effort traffic.

An IETF or SDN control plane can be used to signal and establish guaranteed-rate virtual circuit-switched connections in the *Enhanced-Internet*. The IETF has recently proposed a standardized Internet control-plane protocol, where applications can create end-to-end DiffServ connections using software commands, in RFCs 5974 and 5977 [32–35]. The connections have nominal (i.e., best-effort) QoS metrics, such as bandwidth, latency and burstiness. Alternatively, the *Software Defined Networks* (SDN) technology can also be used to implement a user-programmable control plane [27]. (These technologies do not address the issue of how to establish deterministic connections in the Internet, as there has not previously been a technology to support perfect deterministic virtual circuit-switching in a packet-switched network with 100% throughput.) Each router in the proposed *Enhanced-Internet* can then use the QoS-aware router scheduling algorithm described in [28] to realize the smooth guaranteed-rate connections, while achieving 100% throughput.

The traffic flows between data-centers are aggregated onto optical virtual circuit-switched connections between data-centers, to achieve ultra-low latencies with near-perfect 95–100% link utilizations and improved energy-efficiencies. Each Internet data-center can use a token-bucket-based *Traffic Shaper Queue* (TSQ) in an outgoing access router to aggregate several bursty ON/OFF inter-processor communication streams into a single low-jitter stream (i.e., smooth stream) before transmission onto the high-capacity connections.

5.1 Design of an OEO Single-Chip Lean Router

Each *Enhanced-Internet* router buffers less than 1 packet on average per circuit-switched connection per router [28], typically 1,000 times less buffering than required in traditional *Best-Effort* Internet routers. By Little's Law, the maximum expected queueing delay per router in a circuit-switched connection is typically 1,000 times lower than in packet-switched Best-Effort Internet routers. For example, using a maximum-size IP packet of ≈ 1500 bytes and a guaranteed rate of 100 Gbps, the expected queueing delay in one router is 120 nanosec, and the expected queueing delay along an end-to-end path of 10 routers is ≤ 1.2 microsec. The total end-to-end delay along a circuit-switched connection is essentially equal to the propagation delay of the fiber, as the router queuing delays are reduced to negligible values.

Using 2014 technology, a digital switch for an 20 Tbps lean router can easily be realized on a single CMOS/VCSEL substrate. Consider an 8x8 router with eight 2.5 Tbps IO links. We use the same CMOS/VCSEL design methodology described in [21], which presented the design and analysis of a single-chip 5.12 Tbps CMOS/VCSEL switch using 180 nm CMOS. An $N \times N$ crossbar switch with W -bit wide datapaths constructed with demultiplexers and multiplexers, requires $2WN^2$ binary multiplexer or demultiplexer logic block standard cells. Using a 2 GHz CMOS substrate, an 8x8 crossbar switch with 1,250-bit wide data-paths achieves 2.5 Tbps per port and requires less than 16M transistors. Let the switch support 512 virtual circuit-switched traffic flows between pairs of data-centers. Let the buffering per switch be 1,024 packets (2 per flow). Using the maximum-size IP packets of 1500 bytes (12,000 bits), the switch requires ≈ 12.3 Mbits of high-speed memory. Each D flip-flop (i.e., 1 bit latch) requires 8 transistors. The total transistor count is below 200M, well below the upper limit of CMOS integration in 2014. The Intel Knight's Landing chip uses 5 Billion transistors and consumes 200 Watts. Comparing the transistor counts of the proposed 20 Tbps switch and the Knight's Landing chip, and assuming similar toggle rates, the digital logic for the crossbar switch and buffers in the lean router is estimated to use about 6 Watts.

Consider the 5.12 Tbps single-chip photonic switch described in [21], designed with 180 nm CMOS. The photonic switch has an optical IO bandwidth of 5.12 Tbps, and consumes about 80 Watts. Using the 14 nm CMOS technology, the reduction in CMOS area is given by a scaling factor of $(14/180)^2 = 0.006$. The reduction in switching capacitance is also given by the same scaling factor, so the same 5.12 Tbps digital switch using 14 nm CMOS should require about 0.5 watts, based upon traditional CMOS scaling rules. Using this technique, a 20 Tbps digital switch will use about 2 watts. We will use the more conservative estimate, where a 20 Tbps digital switch uses about 6 watts.

In Canada, we have previously undertaken the design and construction of a multi-Tbps intelligent optical backplane for computing, using the CMOS/VCSEL technology. The issue of bit errors on multi-terabit networks is a potential concern for large scale computing [22]. To avoid excessive bit errors, a 20 Tbps photonic switch should have a very low BER on the optical links,

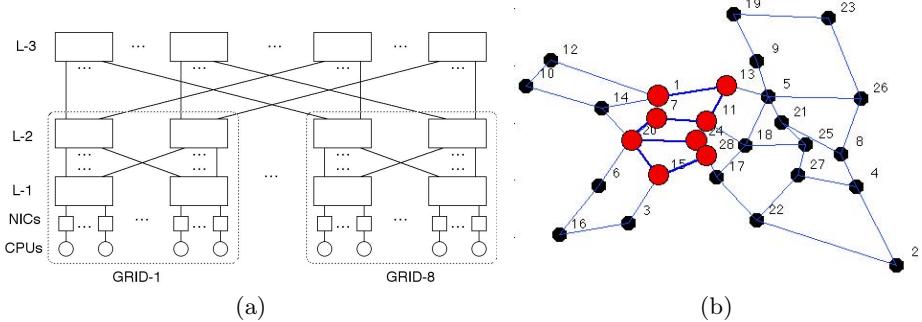


Fig. 4. (a) Global+local communications between data-centers. (b) The Nobel-EU IP network with 28 nodes and 82 directed edges, with 8 data-centers shown by bold dots (in locations: Amsterdam, Brussels, Paris, Lyon, Hamburg, Frankfurt, Strasbourg, Zurich).

i.e., potentially $\leq 10^{-20}$, which can be designed into the CMOS logic using FEC codes [22]. Assuming bit-errors are random events due to thermal noise, the mean time to interruption (MTTI) due to undetected bit-errors when operating at 20 Tbps is $\geq 10^{10}$ seconds, or $\geq 10^5$ days. When multiple 20 Tbps optical switches are operated in parallel, the MTTI decreases proportion to the number of parallel switches. If 1000 switches are operated in parallel, the MTTI is ≥ 100 days. This MTTI is still well above the US DOE design goals of 1 day of continuous operation in 2015.

6 Exascale Computing Analysis

Consider a 100 PF/sec computing system consisting of several smaller HPC grids, as shown in Fig. 4a. Each grid is realized within a data-center, and let there be $DC = 8$ data-centers in Fig. 4a. Each data-center uses an optical network with 2 levels of OEO switches, called L-1 and L-2. The data-center network can be a folded Clos network or a Fat-Tree network, or any other network. Let there be a total of P processors each with a performance of 3 TF/sec. Fig. 4b illustrates the Nobel-EU IP backbone network, where the locations of 8 hypothetical data-centers are shown with bolder nodes, interconnected by bolder edges.

Let N be the number of elements in a Fast Fourier Transform with complex numbers. A butterfly data-flow graph for $N=16$ FFT is shown in Fig. 5a, where each node represents about 8 flops of computation. The graph can be re-arranged to minimize data-transfers between data-centers, as shown in Fig. 5b. The revised FFT graph consists of 2 stages of smaller FFT graphs of size $\sqrt{N} \times \sqrt{N}$ called *factors*, where a type of perfect shuffle permutation interconnects the 2 stages. Each stage has \sqrt{N} factors.

Each factor requires about $8\sqrt{N} \log(\sqrt{N})$ flops. Each factor in the first stage computes \sqrt{N} double-precision complex numbers (with 8 bytes) for transmission to \sqrt{N} factors in the second stage. Each data-center has P/DC processors, and

performs \sqrt{N}/DC factors of computation in each stage. Let each data-center perform its computations in waves consisting of W factors, where $W=1,024$. After each wave of computation, each data-center generates W complex numbers to be transmitted to each factor in the 2nd stage. These numbers are sent as parallel messages (with 8 Kbytes), where each data-center communicates with every other data-center, performing 'All-to-All' communications.

Messages are sent as soon as they are computed, and communications are pipelined with computations. After each data-center has computed all its waves in the first stage, it has received enough messages to start computing the second stage of computations.

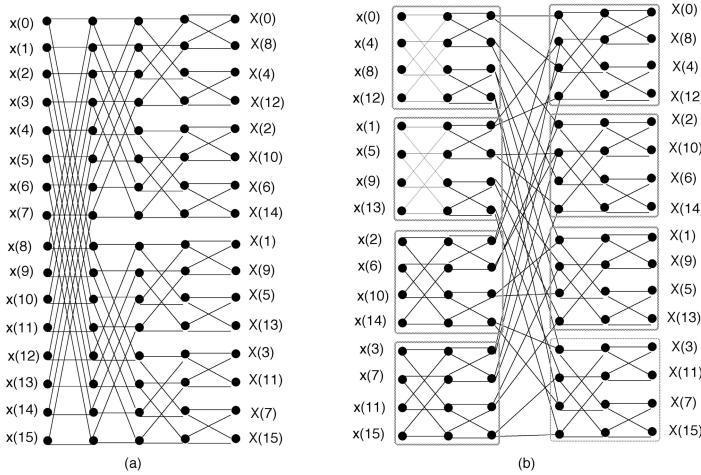


Fig. 5. FFT data-flow graph. (a) Regular, (b) Re-arranged.

The number of flops per wave is about $8W\sqrt{N}\log(\sqrt{N})$. Let Γ denote the Flops/sec of a Knight's Landing processor ($\Gamma = 3 \times 10^{12}$ F/sec). Assuming a 63% efficiency for the processors (the same as the Cray Titan), the time T to perform one wave of computation is

$$T = 8W\sqrt{N}\log(\sqrt{N}) / ((P/DC) \cdot 0.63 \cdot \Gamma) \quad (1)$$

After each wave, each data-center has generated $W\sqrt{N}$ complex numbers requiring $D = 16W\sqrt{N}$ bytes. The transmission-rate R from each data-center, measured in bytes per second, is therefore

$$R = D/T \approx (\log(\sqrt{N}) / ((P/DC) \cdot 0.63 \cdot \Gamma))^{-1} \quad (2)$$

Letting $N = 2^{48}$, then $\sqrt{N} = 2^{24} = 16M$. It takes about 416 nanosec for a data-center to compute 1 factor, about 426 microsec to compute one wave, and about 0.87 seconds for the system to compute the first stage of factors. After

the stage 1 computations, the stage 2 computations can start. In stage 1, each data-center will complete 2K waves of computation and send 2K messages to each factor in stage 2. Assuming a global networking latency of ≤ 5 millisecond (in layer L-3 of Fig. 4a), then each data-center can start computing waves in stage 2 immediately after stage 1 finishes, as the results of most waves of computation in stage 1 have already been received by each data-center.

Each data-center has an optical bandwidth of $R \approx 4,693$ Tbps in the layer-3 switches (L-3). Our estimate is larger than the DOE design target of 20 GB/sec per processor or $R = 655$ Tbps per data-center [3], so we will proceed with the DOE design target. In Fig. 4a, the top layer of switches (L-3) will process $8R = 5.24$ Pbps, and requires about 250 photonic ICs (at 20 Tbps each). Layers L-1 and L-2 each process twice this amount of optical data, and each require about 500 photonic ICs. The *Network Interface Cards* (NICs) originate and terminate 5.24 Pbps, and require the equivalent of 250 photonic ICs (where the optical IO is now distributed over several smaller NICs). In total, a localized 100 PF/sec HPC grid system as shown in Fig. 4a will require about 1,500 photonic ICs. The power required for 1,500 photonic ICs is 0.24 MW, which is increased by 30% to 0.31 MW to account for cooling.

In a localized 100 PF/sec HPC grid system as shown in Fig. 4a, the total power used for optical communications is ≈ 0.31 MW, and the power used for computations is ≈ 13.1 MW. The ratio of power used for communications vs. computations is about 2%.

Using 2015 technology, each photonic IC can easily support an optical packet switch with 20 Tbps. Due to Moore's Law, the power used for the CMOS optical transceivers will decrease with time, and the optical IO bandwidth will increase with time, at least over the next several years. The cost of the CMOS substrates for the photonic switches will be very low, since the CMOS die area is small. If each single-chip photonic switch can be developed at a cost of \$50,000, then the cost for all the photonic switches to enable cloud computing at this scale is about \$75 Million. These costs will decrease with time over the next several years, due to Moore's Law.

7 The Feasability of Exascale Cloud Computing

There are thousands of data-centers world-wide, and most operate at light loads (50% or less). According to Roomey [12, 13], global data-centers consumed about 157.5 billion KW-hrs of energy in 2005. The annual operating expenses for data-center energy is about \$11 Billion US. Given the low utilizations, the energy costs of data-center inefficiencies exceed \$5.5 Billion US per year. If these global data-centers can be utilized to perform useful work such as cloud computing, these energy costs (\$5.5 Billion/US per year) can be recovered.

The European-Union has the highest density of Internet transmission capacity in the world, it should have the highest density of data-centers in the world, and the average distance between cities in the Nobel-EU topology is a few hundred kilometers, resulting in very small fiber latencies. For the HPC cloud solution, we

assume 8 data-centers at these locations are used: Amsterdam, Brussels, Paris, Lyon, Hamburg, Frankfurt, Strasburg, and Zurich, as shown on Fig. 4b.

In the cloud solution, the top layer of switches L-3 in Fig. 4a is distributed over the 8 nodes highlighted in the Nobel-EU network in Fig. 4b. Each connection between data-centers traverses about 2.5 IP links on average. Each IP link between data-centers is therefore provisioned with 2.5 times the bandwidth per data-center (655 Tbps). The relevant 8 nodes in the Nobel-EU are interconnected with 10 edges. The aggregate optical bandwidth needed in L-3 switches to support the cloud communications is about 13.1 Pbps, which will require about 655 photonic ICs.

In the cloud solution, the DSDM optical communications over multi-mode fiber within a data-center must be transformed to DWDM communications over single mode fiber between data-centers. Assume Fujitsu DWDM Flashwave 9500 optical crossconnect and transponders are used in the IP physical layer. Each crossconnect supports 88 DWDM channels at 100 Gbps for a peak bandwidth of 8.8 Tbps, and consumes 800 watts (www.fujitsu.com). Therefore, the power for physical layer DWDM optical communications is ≈ 91 watts per Tbps, or ≈ 91 pJ/bit. Using this Fujitsu technology, the total power needed for the optical physical layer communications between data-centers is ≈ 1.2 MW.

The cloud solution incurs the extra capital cost and energy costs of the physical layer DWDM components. The cloud solution also incurs the extra capital and energy costs of a slightly larger number of OEO photonic switches used in the 8 nodes of the Nobel-EU topology (655 vs. 250). However, the cloud solution removes the capital costs of building a dedicated 100 PF/sec HPC grid computer and associated power supply, typically in the hundreds of millions of dollars, by using existing under-utilized data-centers.

By using the ultra-low latency lean OEO routers, the communication delays between remote data-centers is reduced to the fiber latency, which is typically ≤ 5 millisec in the Nobel-EU topology, and multithreading can now be used to hide these latencies, resulting in improved utilization and energy-efficiency in the data-centers. In the HPC cloud solution, data-centers which were previously under-utilized can now be fully utilized, resulting in higher efficiency. The key enabling technology is the low-cost low-latency OEO routers. If the OEO routers can be manufactured as moderate cost commodities, as industry demands for higher bandwidth communications grow, large-scale HPC cloud computing could become prevalent.

7.1 Bisection Bandwidth

The optical fiber cables used in continental networks typically contain 100s of fibers, since the cost of installing the cables is very high. Typically very few fibers are activated, and the un-used fiber is called "dark fiber". The aggregate bandwidth of any IP network can be increased significantly by activating dark fiber, which already exists as an un-exploited capital cost. Referring to Fig. 4b, if each edge contains 200 fibers at 50 Tbps each, then the bandwidth per edge can approach 10,000 Tbps. By exploiting Cloud Computing and exploiting existing

under-utilized Internet data-centers, the upfront capital costs of building a dedicated HPC grid machine can be removed, and the cost savings can be invested into increasing the aggregate bandwidth of the IP network. The cloud computing service would be available as a shared resource on a pay-for-use model without excessive front-end capital costs, and its cost will be distributed over numerous industries, including industrial HPC, the pharmaceutical and computational biology/genomics industry, gaming, and entertainment.

An entity like the US DOE typically already has several geographically distributed HPC grid computers such as the Cray Titan. One attractive option to achieve Cloud Computing is to link these existing highly-efficient HPC grid machines over the cloud using the newer communications technologies described in this paper. This approach retains the use and security of highly-efficient distributed HPC grid machines, while achieving a virtual machine with the aggregated computing power of all HPC grids combined.

8 Conclusions

This paper explores the impact of new technologies on Exascale Grid/Cloud computing systems and data-centers, including; (i) new low-power multi-core CMOS processor technologies, (ii) new Photonic integrated circuit technologies, and (iii) a new 'lean' internet router design which supports ultra-low latency virtual circuit-switched communications between data-centers. An entire lean router can be realized on one photonic IC, which unlocks the power of integrated photonics. By exploiting ultra-low latency optical connections, high bisection-bandwidth and multithreading, the utilization and energy-efficiency of HPC grid and cloud computing systems will improve. Using 2015 technologies, we have presented the high-level predictive designs of 100 PF/sec HPC Grid and Cloud systems, comprising 8 small data-centers each supporting 4K nodes (Knight's Landing processors) each with 20 GB/sec of node interconnect bandwidth, focussing on the global networking. The Grid design has a peak theoretical performance of ≈ 100 PF/sec, a sustained performance of ≈ 62 PF/sec, and consumes ≈ 13.1 MW for computations, and ≈ 0.31 MW for communications. The Cloud design uses ≈ 1.5 MW for communications. In both designs, the energy used for communications has been reduced by 2 or 3 orders of magnitude, relative to current state of the art routers (i.e., 30 pJ/bit or 130 pJ/bit vs. 10 nJ/bit). These performance figures will continue to improve as Moore's Law of technology scaling holds, until the fundamental limits of CMOS scaling are reached.

References

1. Dongara, J., et al.: The International Exascale Software Project Roadmap, <http://www.exascale.org/mediawiki/images/2/20/IESP-roadmap.pdf>
2. US Dept. of Energy, Advanced Scientific Computing Advisory Committee Subcommittee on Exascale Computing, The Opportunities and Challenges of Exascale Computing, Fall (2010), <http://science.energy.gov/>

3. US Dept. of Energy, Exascale Initiative Steering Computing, SOS 14: The Challenges in Exascale Computing (Slides),
http://www.csm.ornl.gov/work-shops/SOS14/documents/dosanjh_pres.pdf
4. Yelik, K., Coghlani, S., Draney, B., Canon, R.S.: Magellan Report on Cloud Computing for Science. US Dept. of Energy, Office of Science (December 2011),
<http://science.energy.gov/>
5. Shalf, J., Dosanjh, S., Morrison, J.: Exascale Computing Technology Challenges. In: Palma, J.M.L.M., Daydé, M., Marques, O., Lopes, J.C. (eds.) VECPAR 2010. LNCS, vol. 6449, pp. 1–25. Springer, Heidelberg (2011)
6. Shalf, J., Kamil, S., Oliker, L., Skinner, D.: Analyzing Ultra-Scale Application Communication Requirements for a Reconfigurable Hybrid Interconnect. In: Proc. 2005 ACM/IEEE Conf. on Supercomputing (2005)
7. Kendrick, J.: Cloud Computing Not Quite Ready for the Labs: US Government Report, Forbes (January 19, 2012), <http://www.forbes.com>
8. Gevros, P., Crowcroft, J., Kerstein, P., Bhatti, S.: Congestion Control Mechanisms and the Best-Effort Service Model. IEEE Network Mag. (May/June 2001)
9. Iyer, S., Komella, R.R., McKeown, N.: Designing Packet Buffers for Router Linecards. IEEE Trans. Networking 16(3) (June 2008)
10. Bolla, R., Bruschi, R., Davoli, F., Cucchietti, F., et al.: Energy Efficiency in the Future Internet: A Survey of Existing Approaches and Trends in Energy-Aware Fixed Network Infrastructures. IEEE Comm. Surveys and Tutorials, 2Q (2011)
11. Raghavan, B., Ma, J.: The Energy and Emergy of the Internet, Hotnets (2011)
12. Masanet, E., Brown, R.E., Shehabi, A., Roomey, J.G., Nordman, B.: Estimating the Energy Use and Efficiency Potential of U.S. Data-Centers. Proc. IEEE (2011)
13. Roomey, J.G.: Worldwide Electricity Used in Data-Centers, Environ. Res. Letters, IOP Science (2008)
14. Tucker, R.S., Parthiban, R., Baliga, J., Hinton, K., Ayre, R.W.A., Sorin, W.V.: Evolution of WDM Optical IP Networks: A Cost and Energy Perspective. OSA JLT 27(3) (2009)
15. Ben Yoo, S.J.: Energy Efficiency in the Future Internet: The Role of Optical Packet Switching and Optical Label Switching. IEEE JSTQE 17(2) (March/April 2011)
16. Baliga, J., Ayre, R.W.A., Hinton, K., Tucker, R.S.: Green Cloud Computing: Balancing Energy in Processing, Storage and Transport. Proc. IEEE 99(1) (January 2011)
17. Greentouch White Paper, Greentouch Green Meter Research Study: Reducing Net Energy Consumption in Communications Networks by up to 90% by 2020 (June 2013), <http://www.greentouch.org>
18. Kantor, D.: Knights Landing Details (January 2, 2014),
<http://www.realworldtech.com>
19. Anantharam, V., McKeown, N., Mekittikul, A., Walrand, J.: Achieving 100% Throughput in an Input Queued Switch. Trans. Comm. 47(8) (1999)
20. Keslassy, I., Kodialam, M., Lakshman, T.V., Stilliadis, D.: On Guaranteed Smooth Scheduling for Input-Queued Switches. IEEE/ACM Trans. Networking 13(6) (December 2005)
21. Szymanski, T.H., Wu, H., Gourgy, A.: Power Minimization in Multiplexer-Based Optoelectronic Crossbar Switches. IEEE Trans. VLSI 13(5) (May 2005)
22. Szymanski, T.H., Tyan, V.: Error and Flow Control for a Terabit Free-Space Optical Backplane. IEEE JSTQE (March/April 1999 1999)
23. Lin, C.K., Tandon, A., Djordjev, K., Corzine, S.W., Tan, M.R.: High-Speed 985 nm Bottom-Emitting VCSEL Arrays for Chip-to-Chip Parallel Optical Interconnects. IEEE JSTQE 13(5) (2007)

24. Doany, F.E., Benjamin, G.L., Daniel, M.K., Rylyakov, A.V., Baks, C., Jahnes, C., Libsch, F., Schow, C.L.: Terabit/Sec VCSEL-Based 48-Channel Optical Module Based on Holey CMOS Transceiver IC. *IEEE JLT* 31(4) (2013)
25. Arakawa, Y., Nakamura, N., Urino, Y., Fujita, T.: Silicon Photonics for Next Generation System Integration Platform. *IEEE Comm. Mag.* 51(3) (2013)
26. Vlasov, Y.A.: Silicon-CMOS Integrated Nano-Photonics for Computer and Data-Communications Beyond 100G. *IEEE Comm. Mag.* (February 2012)
27. Kim, H., Feamster, N.: Improving Network Management with Software Defined Networking. *IEEE Comm. Mag.* 5(2) (2013)
28. Szymanski, T.H.: Max-Flow Min-Cost Routing in a Future Internet with Improved QoS Guarantees. *IEEE Trans. Comm* 61(4) (April 2013)
29. Szymanski, T.H.: Methods to Achieve Bounded Buffer Sizes and Quality of Service Guarantees in the Internet Network. US Patent (March 2014)
30. Szymanski, T.H.: An Ultra-Low Latency Energy-Efficient Internet for Cloud Services. *IEEE Transactions on Networking* (submitted)
31. IETF RFC 5290, Comments on the Usefulness of Simple Best Effort Traffic (July 2008)
32. IETF RFC 5865, A Differentiated Services Code Point (DSCP) for Capacity-Admitted Traffic (May 2010)
33. IETF RFC 5974, NSIS Signaling Layer Protocol (NSLP) for Quality-of-Service Signaling (October 2010)
34. IETF RFC 5976, Y.1541-QOSM: Model for Networks Using Y.1541 Quality-of-Service Classes (October 2010)
35. IETF RFC 5977, RMD-QOSM: The NSIS Quality-of-Service Model for Resource Management in Diffserv (October 2010), <http://www.ietf.org>

SADDLE: A Modular Design Automation Framework for Cluster Supercomputers and Data Centres

Konstantin S. Solnushkin

Saint Petersburg State Polytechnic University, Saint Petersburg, Russia

konstantin@solnushkin.org

<http://ClusterDesign.org/saddle>

Abstract. In this paper we present SADDLE, a modular framework for automated design of cluster supercomputers and data centres. In contrast with commonly used approaches that operate on logic gate level (Verilog, VHDL) or board level (such as EDA tools), SADDLE works at a much higher level of abstraction: its building blocks are ready-made servers, network switches, power supply systems and so on. Modular approach provides the potential to include low-level tools as elements of SADDLE's design workflow, moving towards the goal of electronic system level (ESL) design automation. Designs produced by SADDLE include project documentation items such as bills of materials and wiring diagrams, providing a formal specification of a computer system and streamlining assembly operations.

Keywords: Design, Automation, CAD, EDA, Cluster, Supercomputer, Data centre.

1 Introduction and Motivation

As of today, design automation in electronics mainly concerns using languages such as Verilog and VHDL to create devices from logic gates, or using electronic design automation (EDA) tools to create individual boards. Both approaches are well-developed, mostly because automation is indispensable in these fields. Given the current number of transistors on a chip or components on a board and the complex nature of interrelations between characteristics of systems under design, the use of computer-aided design (CAD) tools is a requirement, not a whim. Additionally, manufacture of modern electronics requires the use of industrial robots, which implies the necessity of having precise documentation that is best produced automatically with CAD tools.

On the other hand, in the field of cluster supercomputer design and data centre design in general, automation has not yet found a widespread use. An often quoted reason is that most tasks can be easily handled by a human designer, hence corresponding CAD tools do not exist because they are not required. With the size of supercomputers and data centres continuously growing, manually generated designs may no longer be cost-optimal. Common tasks that designers need

to solve are: choosing types and the number of components in a server, calculating the number of servers and switches based on performance requirements, placing equipment in racks, etc. The vast size of design space justifies the use of automation.

Currently, both cluster supercomputers and warehouse-scale data centres tend to be created with identical building blocks. Deviations from established practice, if any, are infrequent and limited. We argue that this regrettable situation stems from the absence of appropriate CAD tools: when consequences of design choices are hard to predict, human designers tend to constrain the variety of their designs due to bias, personal preferences or lack of time, using familiar components in standard configurations. However, this may lead to under-exploration of the design space and thus to non-optimal designs.

Design automation tools that operate at logic gate level are still called, through inertia, “high-level synthesis” tools, but, compared to state-of-the-art needs, they turn out to be very low-level. The research community has called for an overarching electronic system level (ESL) design automation approach [1]. SADDLE is the response to this challenge. It operates on a very high level (its building blocks are servers and switches), serving as a system-level complement to existing low-level EDA tools.

SADDLE is a modular CAD tool that allows a designer to quickly evaluate multiple design choices in terms of their technical and economic characteristics, conduct “what-if” scenario analyses, and automatically obtain comprehensible design documentation to simplify assembly process. The acronym stands for “Supercomputer and data centre design language”.

2 Related Work

The problem of selecting an optimal configuration of a computer system has been addressed previously. One of the earliest works was *R1*, an expert system created by John P. McDermott in late 1970s [2]. Its main task was to configure VAX-11/780 minicomputers made by Digital Equipment Corporation. The design space was large due to an assortment of available peripheral devices; there were also various mechanical and power constraints that were taken into account. *R1* was a production rule expert system which operated on a set of 480 rules representing domain knowledge. It could produce detailed assembly documentation including floor plans and cable wiring tables, and therefore set the standard for future automated configurers of computers.

In 1998, Pao-Ann Hsiung et al. [3] proposed *ICOS*, an Intelligent Concurrent Object-Oriented Synthesis methodology which focused on design of multiprocessor systems. According to the object-oriented approach, system components are modelled as classes with hierarchical relationships between them. Previously synthesised subsystems can be reused as building blocks of new designs; machine learning and fuzzy logic are used to determine feasibility of the reuse.

In 2005, William R. Dieter and Henry G. Dietz published a technical report [4] detailing their methodology called *Cluster Design Rules (CDR)*, as well

as patterns that emerged through the continuous use of the *CDR* tool. This methodology is perhaps the first attempt aimed specifically at designing cluster supercomputers. Designs were evaluated using a weighted linear combination of metrics. Although *CDR* did not turn into a comprehensive product, it was a successful project that pointed to new directions for research in its field.

Among the most important observations in the practical use of the *CDR* tool was that the global optimality of a supercomputer design cannot be inferred from local optimality of any of its components: for example, using a CPU with the lowest price to performance ratio does not lead to a supercomputer design with the lowest price at a given performance. This justifies thorough automated inspection of the design space.

Nagarajan Venkateswaran et al. [5] addressed the problem of automated design again in 2009. Their methodology, “*Modeling and integrated design automation of supercomputers (MIDAS)*”, is aimed at Supercomputers-on-a-Chip (SCOC), but can be generalised to wider areas as well. With *MIDAS*, supercomputers-on-a-chip are built using dedicated IP cores implementing specific algorithms. The number of cores of each type to be placed on a chip is determined using performance modelling and simulation. Simulated annealing is then used to select optimal configurations. *MIDAS* does not address problems of building large multi-node cluster supercomputers, but serves as an important link for implementing Electronic System Level (ESL) design approach: from chips to servers to supercomputers.

Design of large-scale computers was addressed once again by Barroso et al. in 2013 [6]. However, their work mostly considers platforms for generic data centre workloads rather than high-performance computing.

SADDLE is different from the previous work in that it provides its users with a practical and useful tool for solving design problems. Its extensibility is guaranteed by modularity, allowing to interface with external design tools. Special emphasis is given to accurate estimation of economic characteristics, which is a requirement for producing cost-effective designs.

Additionally, we note that the present paper expands upon ideas previously proposed in research posters [7] and [8].

3 Design Flow

Design of data centres is akin to that of cluster supercomputers, with the difference that data centres often have many smaller groups of servers, each suited for a particular task. In other respects, design for both types of installations is based on the same principles.

In SADDLE, design flow is modelled after actions of human designers. Conceptually, there are four stages: (1) exploration of compute node configurations for selection of promising candidate solutions; (2) choice of appropriate number of compute nodes in selected configuration to satisfy performance constraints; (3) automated elaboration of design decisions for infrastructural systems (network, power supply system, etc.); (4) equipment placement, floor planning and documentation generation.

From the point of view of SADDLE’s user, the design procedure basically answers the question, “What configuration of a computer has the lowest cost at a given performance?”. Configuration is defined by instances of components used to build the system and structure of their connections with each other. The detailed sequence of design steps is given below:

1. Form a set of configurations of a compute node;
2. Weed out unpromising configurations using constraints and heuristics;
3. For each configuration in the pool of candidate solutions, repeat:
 - (a) Using inverse performance modelling, calculate the number of compute nodes required to satisfy performance constraints
 - (b) Design an interconnection network
 - (c) Design infrastructural systems (e.g., a power supply system)
 - (d) Place equipment into racks, locate racks on the floor, route cables
 - (e) Choose the best design automatically (using criterion functions) or manually
 - (f) Generate documentation

Steps 1 and 2 can be repeated for as many models of servers as a designer wishes to consider, forming a bigger pool of candidate solutions for the next steps; e.g., blade servers and regular rack-mount servers can be inspected side by side, and results compared to choose the best alternative.

On every stage, semi-complete designs can be checked against a set of constraints specified by the designer; these can be physical constraints stipulated by available machine room space, power, etc., or budgetary constraints imposed on capital or operating costs. Violating configurations can be immediately discarded; this way they don’t participate in further design steps which would lengthen the total time to solution.

SADDLE is modular; some of the modules are implemented as subroutine calls while others can be queried over the network.

4 Representation of Configurations

Building blocks in SADDLE are compute nodes. Their configurations are represented with directed acyclic multipartite graphs: partitions correspond to functions, vertices denote components or, more generally, possible implementations of each function, while edges represent compatibility of components. Each path in the graph represents a valid configuration of a technical system, in this case, a compute node. While the use of graph theory for representing configurations is not new, a major advancement comes from assigning to vertices sets of expressions which are evaluated during graph traversal.

For example, the graph in Figure 1 represents a dual-CPU compute node. Vertices from the database are instantiated in the graph as many times as necessary. Graph traversal yields eight paths, each corresponding to a valid configuration. Arrows on the edges are not shown to reduce clutter since traversal is performed in a natural start-to-end pattern.

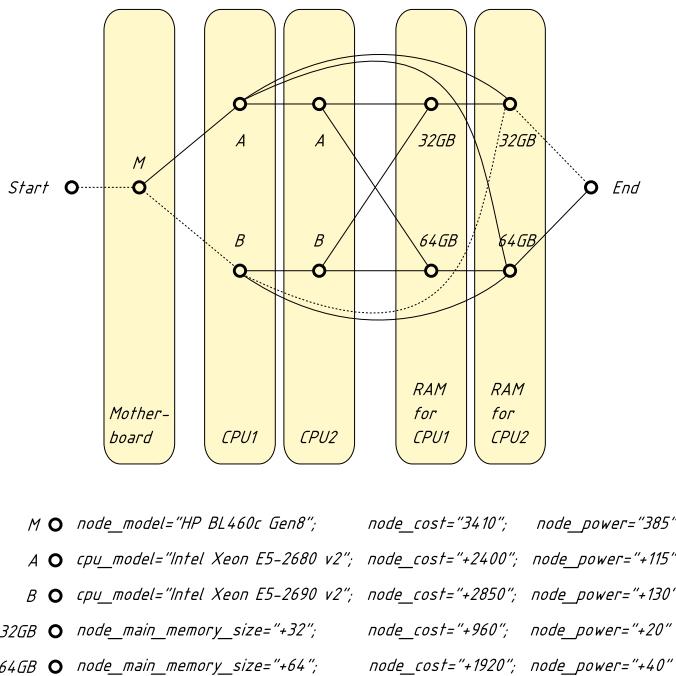


Fig. 1. Graph representation of eight configurations of a dual-CPU compute node

One of the paths is highlighted with dotted lines. It includes vertices “M”, “B” and “32GB”; in other words, the corresponding configuration includes a motherboard, one “Intel Xeon E5-2690 v2” CPU and 32 GBytes of memory. Every time a vertex is traversed, expressions prescribed to it are evaluated; for example, expression `node_cost="+2850"` for vertex “B” means to add a specified number to the existing value of this metric, or to zero if no current value exists. In this case, by the time the “End” vertex is reached, metric `node_cost` will receive a value of “7220”; that is the cost of this configuration of the compute node.

Virtually any technical or economic metric can be calculated in a similar manner. For example, we calculate the number of CPU cores per node, peak floating-point performance, power consumption and weight of a node, amount of RAM per core and many other metrics.

For brevity, we will omit details of expression grammar and graph transformations. We note, however, that directed acyclic graphs, when used for representing configurations, have a somewhat lower expressive power than undirected graphs with cycles, yet their visual comprehension is easier.

Graphs representing configurations of currently produced servers with real-world complexity tend to generate from 50 to 250 configurations. Most of them will not lead to good designs of cluster supercomputers and therefore can be discarded. In SADDLE, there are two mechanisms to do this: constraints and

heuristics. Constraints can be specified on any metrics of a configuration. For example, the user can request to discard configurations that have CPUs with too few or too many cores, or where the amount of memory per core is too small, or configurations that don't have a specific type of network adapter.

Constraints allow to quickly reduce the size of design space, dramatically decreasing overall time to solution. However, the user should be careful not to prune too aggressively, because intuition cannot serve as a substitution for exhaustive search. For example, the user may be tempted to discard CPUs with the lower number of cores, only to find later that they actually had a larger amount of cache memory per core and would have delivered better performance for a customer's application.

As also mentioned in section 3, constraints can be imposed at any stage of the design process; this way, even if suboptimal configurations were not discarded immediately after generation, they can still be weeded out at later stages.

The second mechanism to deal with combinatorial explosion is heuristics. Already calculated metrics can be arbitrarily combined to form a simple objective function, and configurations can be discarded based on the value of this function. For example, we found that using the ratio of compute node's cost to its peak floating-point performance as a predictor of quality allows to safely discard 80% of configurations.

5 Performance Modelling

The next design step involves calculation of the number of compute nodes required to reach a specified performance goal. In general, performance cannot be inferred from peak floating-point performance that is calculated when generating configurations; this is especially true for workloads that are not floating-point oriented. Instead, SADDLE calls design modules that implement performance models.

A performance model generally accepts on its input a number of metrics of a single compute node as well as the number of nodes (or other “computing blocks”: cores, CPUs, accelerator boards, racks, etc.), and outputs projected performance of a cluster supercomputer. We call such models “direct”. We also introduce the notion of *inverse* performance models; these accept metrics of a compute node and desired performance, and output the number of compute nodes required to achieve the specified performance level.

Inverse performance modelling is a specific type of an inverse problem. We solve the problem in two stages: in a forward pass, we repetitively call a corresponding direct performance model with monotonically increasing the number of nodes (say, in powers of 2). When the specified performance goal has been overreached, we use bisection method to determine a more exact number of “computing blocks”.

For demonstration purposes, we implemented a simple analytical performance model of computational fluid dynamics (CFD) software “ANSYS/Fluent”; it's fast although not very precise. It predicts performance based on CPU clock frequency, the number of CPU cores in a cluster and the type of interconnection

network (InfiniBand or 10Gbit Ethernet). Inverse performance model is implemented in the same module and calls direct performance model according to the above algorithm. We found that only a few calls to the direct model are needed to identify with sufficient precision the number of CPU cores that a cluster computer is required to have.

Generally, performance models are not limited to analytic, they can be of any nature, including simulators or FPGA prototypes, although these require more time and resources for direct and especially inverse performance modelling. This generality can be used to plug-in the results of low-level design workflow into SADDLE's high-level workflow.

Indeed, suppose there is a new CPU under development, and there exists a simulation framework that can predict performance of a cluster computer based on this type of CPU on certain workloads. Several CPU parameters can be tweaked (number of cores, sizes of caches), and each valid combination represents a separate model of this future CPU. Existing EDA tools can be used to calculate performance and power consumption of each model, with tools such as CACTI [9] (for memory hierarchies) and Orion [10] (for networks-on-chip) comprising the end-to-end simulation framework. With more effort we can also estimate per-item cost of producing each CPU model when using mass production. This is enough to construct configuration graph from section 4.

Then, a simple design module can be created that queries the aforementioned simulation framework, estimating performance and cost for each cluster configuration when varying CPU model used and the number of CPUs in a cluster. This will allow engineers to use SADDLE to perform directed search of best CPU configurations, and to further build cost-effective cluster configurations. It is important to note that it is not necessarily the cheapest or the fastest CPU that brings optimality to the cluster configuration as a whole, hence the possibility to perform automated directed search is very beneficial.

Using end-to-end simulation will allow to perform comprehensive what-if analysis, answering questions such as “How will adding more cache memory or more floating-point units to a CPU impact performance, power and cost of the whole supercomputer and its infrastructure?”

Compare this to the approach used by Sandia's SST [11], which creates a loop of low-level, fine-grained simulators that feed results into higher-level modules. Our approach generalises that of SST by extending it to the data centre level and by more thoroughly dealing with costs, both capital and operating.

Put another way, existing EDA tools search for a compromise between performance, power and area of an individual chip, while SADDLE searches for the same compromise for a data centre, also adding cost to the mix of metrics.

6 Design of Subsystems

SADDLE can design interconnection networks and power supply systems by querying design modules. Modules are web applications that can also be used standalone through a web browser. Design of other infrastructure such as storage systems can be enabled by creating corresponding modules.

SADDLE queries the network design module to design fat-tree and torus interconnection networks. Module’s database contains monolithic and modular switches, using the same graph representation as detailed in section 4, with cost, power, size and weight figures for each switch configuration. As a result, all essential metrics of interconnection networks can be estimated and integrated into the overall cluster design.

Input parameters for the network design module are the number of compute nodes to be interconnected and the desired topology. The module can use any objective function; different configurations of switches are tried in an attempt to minimise this function, which by default is equipment cost.

Choosing an uninterruptable power supply (UPS) system is done by querying another design module that works very similarly to the above. In this case, input parameters are electric power that must be provided and an optional backup time when running on batteries.

7 Equipment Placement and Floor Planning

Inverse performance model returns the number of servers in the future supercomputer, and individual design modules return the amount of infrastructural equipment such as network switches and uninterruptable power supplies. With this information, the next stage can begin: placing equipment into racks and locating racks on the floor. SADDLE already implements a simple algorithm for these tasks, and more elaborate algorithms can be added as necessary.

Currently, SADDLE uses a set of rules to place equipment into racks. These rules are akin to heuristics proposed by Mudigonda et al. [12], but ours are more general as they allow to place equipment of different sizes. Placement occurs in the following order:

1. Core switches;
2. “Compute blocks”; each block is an edge switch together with compute nodes connected to it
3. UPS equipment

Every of these equipment types can be placed according to one of three strategies: (a) consolidation: equipment items are placed into racks as densely as possible; (b) separation: each item is placed in its own rack; (c) spread: each subsequent item is placed in a rack that is N racks apart from the previous item; the latter allows to “spread” equipment in a machine room.

Core switches are by default placed using separation strategy. Compute blocks are placed using consolidation strategy: compute nodes from a block are added to the first rack that has enough free space, and then the accompanying edge switch is added to the same or a nearby rack. When the current rack cannot fit any more compute blocks, the next rack is used (or created, if necessary).

While edge switches should preferably be placed in the same rack with their compute nodes, they do not have to be physically adjacent. We place edge

switches to the top of the rack in an effort to minimise the length of inter-rack cables, while compute nodes are placed in the bottom of the rack to ensure mechanical stability. Therefore placement results in “gaps” in the middle of the racks, rather than at the top or at the bottom.

The next step is to locate racks on the floor. Yet another design module is used to determine optimal dimensions of the machine room, in terms of the number of rows and racks per row, taking into account rack dimensions and clearances and trying to produce a roughly square shape.

Racks are laid out on the floor in a serpentine pattern; this is intended to ensure that a block of compute nodes in the end of a row is not located too far away from its edge switch (which is placed separately and can in principle be put in the next row).

SADDLE calculates cable lengths using Manhattan distance, prints the list of required cables and the table detailing their connections, and draws front views of racks, with cables routed in overhead trays, in SVG (Scalable Vector Graphics) format.

Cable length can be minimised by optimally locating switches in racks and racks on the floor. For example, Fujiwara et al. [13] formulate rack layout problem as a facility location problem, where the total inter-rack cable length is sought to be minimised, and solve it using simulated annealing, reducing total cable length by 29..40%. Similar algorithms can be added to SADDLE in the future.

8 Experimental Evaluation

To evaluate SADDLE, we used it to design two cluster supercomputers, with peak floating-point performance of 100 TFLOPS and 1 PFLOPS, respectively. First, we prepared a database of compute nodes that defines the configuration graph. We used “BL460c Gen8” blade servers made by Hewlett-Packard, with one or two CPUs and without accelerator boards such as GPGPUs. This graph generates 56 configurations of compute nodes. We imposed constraints to select configurations with InfiniBand adapters, and then used a heuristic, ranking configurations according to the ratio of the cost of individual compute node to its peak floating-point performance and choosing a configuration with the lowest value of this metric.

This turned out to be a configuration with two ten-core Intel Xeon CPUs. We then used this configuration as a building block of our cluster supercomputers. The two designs are compared in Table 1. Operating costs were calculated with the following assumptions: system lifetime is 3 years, electricity price is \$0.35 per kW·hour, and price of stationing one rack in a data centre are \$3,000 per year. Note that prices are retail and therefore do not reflect possible discounts for such large-scale procurements.

“Tomato equivalent” mentioned in the last line of the table refers to the idea of reusing waste heat from the supercomputer for agricultural purposes, such as growing tomatoes in greenhouses. Calculations by Andrews and Pearce [14] indicate that tomato crops could reach roughly 400 kg per 1 MW of reused heat per day.

SADDLE script to produce the designs is given in Figure 2. Seventeen lines of code are enough to make quick conclusions and facilitate more detailed search. The script also highlights possibilities of performing “what-if” analyses: for example, settings such as electricity price, system lifetime, rack height or stationing price, etc. can all be quickly changed and the script re-run to estimate design-wide changes.

Table 1. Comparison of two cluster computer designs, with peak floating-point performance of 100 TFLOPS and 1 PFLOPS

	100 TFLOPS	1 PFLOPS
Compute node model	HP BL460c Gen8	
CPU model	Intel Xeon E5-2680 v2	
CPU clock frequency, GHz	2.8	
Node peak performance, GFLOPS	448	
Node power, W	651	
Number of compute nodes	224	2,233
Number of racks (incl. UPS)	8	72
Floor space size, m ²	24	144
Cable length, m	1,066	26,392
Power, kW	159	1,600
Weight, tonnes	3.3	31.6
Costs, millions US dollars:		
Capital expenditures	3.1	31.9
Operating expenditures	1.6	15.7
Total cost of ownership	4.6	47.5
Tomato equivalent, kg per day	63	630

8.1 Avalanche Changes

SADDLE also makes it easy to explore the avalanche effect of seemingly small changes on the whole design. For example, if blade servers are replaced with ordinary rack-mount servers (even with the same internal components), this immediately changes characteristics such as compute node size and cost. Different network switches then need to be used, and power consumption of the machine may also change, leading to a different configuration of the UPS system.

To explore this scenario, we designed a 1 PFLOPS machine with rack-mount servers from a different manufacturer, keeping node parameters (CPU and RAM) the same. Such compute nodes are 32% cheaper; however, the complete machine built with them is only 18% less expensive in terms of the total cost of ownership, because network and UPS cost do not change significantly. At the same time, this machine, due to lesser density of computing equipment, occupies 100 racks instead of 72, eventually leading to a floor space that is 30% bigger (187 m² versus 144 m²). In other words, when space is not a hard constraint, ordinary servers can be used to reduce the total cost of ownership.

```

# Open the database of configurations from specified files
open_db(['db/hp.xml', 'db/intel-xeon-2600.xml',
         'db/hp-blade-memory.xml', 'db/hp-network.xml',
         'db/hp-bl460c_gen8.xml'])
# Only allow configurations with InfiniBand connectivity
constraint("InfiniBand" in network_tech)
# Use a heuristic to filter out unpromising configurations
metric("node_cost_to_peak_performance = node_cost /
       node_peak_performance")
# Select the best configuration according to the heuristic
select_best("node_cost_to_peak_performance")
# Delete inferior configurations
delete()
# Set the number of nodes for 1 PFLOPS of performance and update
# metrics to calculate the number of cores automatically
metric("nodes = ceil(1000000 / node_peak_performance)")
update_metrics()
# Use peak performance instead of calling a model
performance_module['url']='peak'
# Calculate performance for the said number of nodes
performance()
# Specify explicitly to use blade switches from Hewlett-Packard
metric("network_vendor='hp-blade'")
# Design a network
network()
# Design a UPS system
ups()
# Save designed equipment in a group
add_group('compute')
# Place equipment as densely as possible
place(place_params={'strategy': 'consolidate'})
# Route cables and calculate their length
cables()
# Print technical and economic metrics of the design
print_design()
# Draw and save front view of rows
draw_rows([], 'racks.svg')

```

Fig. 2. SADDLE script to produce sample designs from Table 1

9 Extending SADDLE

SADDLE is implemented on top of the Python ver. 3 programming language; its statements are Python functions. It is *intended* to be modified and extended by its users, and its source code is heavily commented to facilitate this: about 40% of lines are comments. Instead of being a massive piece of software with many rarely used knobs, it allows its user to make quick ad-hoc fixes for unconventional usage scenarios.

For example, operating costs are usually calculated on the assumption of continuously running hardware. If hardware only runs during part of the day, the corresponding line that calculates electricity costs can be found in the source code and edited for this particular run.

Future work is planned to further extend SADDLE. Advancements can be made to improve core functionality, for example, to enable automated design of storage and cooling systems.

Performance models, which are of great importance to SADDLE, can be created in cooperation with the authors of supercomputer software, to guarantee that SADDLE produces reliable performance estimates for various architectures, thereby allowing their fair comparison.

Another important task is to ensure that databases contain current models and prices for all types of hardware: compute nodes, network switches and UPS systems. Open format makes it easy for vendors to compile and publish databases for their hardware as often as required.

References

1. Duranton, M., Yehia, S., De Sutter, B., De Bosschere, K., Cohen, A., Falsafi, B., Gaydadjiev, G., Katevenis, M., Maebe, J., Munk, H., Navarro, N., Ramirez, A., Temam, O., Valero, M.: The HiPEAC vision (2010),
<http://www.hipeac.net/roadmap>
2. McDermott, J.: R1: A rule-based configurer of computer systems. Technical Report CMU-CS-80-119, Carnegie-Mellon University (April 1980)
3. Hsiung, P.A., Chen, C.H., Lee, T.Y., Chen, S.J.: ICOS: An intelligent concurrent object-oriented synthesis methodology for multiprocessor systems. ACM Transactions on Design Automation of Electronic Systems (TODAES) 3(2), 109–135 (1998)
4. Dieter, W.R., Dietz, H.G.: Automatic exploration and characterization of the cluster design space. Technical Report TR-ECE-2005-04-25-01, University of Kentucky, Electrical and Computer Engineering Dept. (April 2005),
<http://www.engr.uky.edu/~dieter/pub/TR-ECE-2005-04-25-01.pdf>
5. Venkateswaran, N., Vasudevan, A., Subramaniam, B., Mukundrajan, R.: Ramnath Sai Sagar, T., Manivannan, M., Murali, S., Krishnan Elangovan, V.: Towards modeling and integrated design automation of supercomputing clusters (MIDAS). Computer Science – Research and Development 24, 1–10 (2009)
doi: 10.1007/s00450-009-0085-5
6. Barroso, L.A., Clidaras, J., Hölzle, U.: The datacenter as a computer: An introduction to the design of warehouse-scale machines. Synthesis Lectures on Computer Architecture 8(3), 1–154 (2013)

7. Solnushkin, K.S.: Combinatorial design of computer clusters. In: Proceedings of the International Supercomputing Conference, ISC 2011 (June 2011)
8. Solnushkin, K.S.: Computer cluster design automation using web services. In: Proceedings of the International Supercomputing Conference, ISC 2012 (June 2012)
9. Muralimanohar, N., Balasubramonian, R., Jouppi, N.: Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 3–14. IEEE Computer Society (2007)
10. Kahng, A.B., Li, B., Peh, L.S., Samadi, K.: Orion 2.0: A fast and accurate NoC power and area model for early-stage design space exploration. In: Proceedings of the Conference on Design, Automation and Test in Europe, European Design and Automation Association, pp. 423–428 (2009)
11. Hendry, G., Rodrigues, A.F.: SST: A simulator for exascale co-design. Technical Report SAND2012-1764C, Sandia National Laboratories (March 2012)
12. Mudigonda, J., Yalagandula, P., Mogul, J.C.: Taming the flying cable monster: A topology design and optimization framework for data-center networks. In: Proceedings of USENIX Annual Technical Conference, ATC 2011 (June 2011)
13. Fujiwara, I., Koibuchi, M., Casanova, H.: Cabinet layout optimization of supercomputer topologies for shorter cable length. In: Proceedings of the 2012 13th International Conference on Parallel and Distributed Computing, Applications and Technologies, pp. 227–232. IEEE Computer Society (2012)
14. Andrews, R., Pearce, J.M.: Environmental and economic assessment of a green-house waste heat exchange. Journal of Cleaner Production 19(13), 1446–1454 (2011)

The SIOX Architecture – Coupling Automatic Monitoring and Optimization of Parallel I/O[★]

Julian M. Kunkel¹, Michaela Zimmer¹, Nathanael Hübbe¹, Alvaro Aguilera²,
Holger Mickler², Xuan Wang³, Andriy Chut³, Thomas Bönisch³,
Jakob Lüttgau¹, Roman Michel¹, and Johann Waging¹

¹ University of Hamburg, Germany

² ZIH Dresden, Germany

³ HLRS Stuttgart, Germany

Abstract. Performance analysis and optimization of high-performance I/O systems is a daunting task. Mainly, this is due to the overwhelmingly complex interplay of the involved hardware and software layers. The Scalable I/O for Extreme Performance (SIOX) project provides a versatile environment for monitoring I/O activities and learning from this information. The goal of SIOX is to automatically suggest and apply performance optimizations, and to assist in locating and diagnosing performance problems.

In this paper, we present the current status of SIOX. Our modular architecture covers instrumentation of POSIX, MPI and other high-level I/O libraries; the monitoring data is recorded asynchronously into a global database, and recorded traces can be visualized. Furthermore, we offer a set of primitive plug-ins with additional features to demonstrate the flexibility of our architecture: A surveyor plug-in to keep track of the observed spatial access patterns; an fadvise plug-in for injecting hints to achieve read-ahead for strided access patterns; and an optimizer plug-in which monitors the performance achieved with different MPI-IO hints, automatically supplying the best known hint-set when no hints were explicitly set. The presentation of the technical status is accompanied by a demonstration of some of these features on our 20 node cluster. In additional experiments, we analyze the overhead for concurrent access, for MPI-IO's 4-levels of access, and for an instrumented climate application.

While our prototype is not yet full-featured, it demonstrates the potential and feasibility of our approach.

Keywords: Parallel I/O, Machine Learning, Performance Optimization.

1 Introduction

I/O systems for high-performance computing (HPC) have grown horizontally to hundreds of servers and include complex tiers of 10,000 HDDs and SSDs. On

[★] We want to express our gratitude to the „Deutsches Zentrum für Luft- und Raumfahrt e.V.“ as responsible project agency and to the „Bundesministerium für Bildung und Forschung“ for the financial support under grant 01 IH 11008 A-C.

the client side, complexity of high-level software layers increases. This leads to a non-trivial interplay between hardware and software layers, and diagnosing it has become a task to challenge even experts. Parameterizing the layers for optimum performance requires intimate knowledge of every hardware and software component, including existing optimization parameters and strategies.

The SIOX Project was initiated to shed light on the interactions, and to offer automatic support for optimizing the HPC-I/O stack. Continually monitoring performance and overhead, an I/O system instrumented with SIOX will autonomously detect problems, inferring advantageous settings such as MPI hints, stripe sizes, and possible interactions between them. In this paper we will present first results obtained with the SIOX prototype.

The contributions of this paper are: 1) a description of our modular architecture for monitoring, analysis and optimization. 2) analysis of the overhead for synthetic benchmarks and a climate model. 3) use-cases demonstrating the benefit of automatic optimization.

This paper is structured as follows: Section 2 sketches the state of the art in I/O performance analysis. The modular architecture and implementation of SIOX is introduced in Section 3. Section 4 describes tools to analyze and visualize instrumented applications. We evaluate the overhead of our prototype in Section 5, and demonstrate the potential of this approach in several scenarios. Finally, ongoing and future work is discussed in Section 6 and conclude our experience with the SIOX prototype in Section 7.

2 Related Work

Efforts to monitor I/O behavior are legion, the latest widely-used exponent being Darshan [1], a lightweight tool to gather statistics on several levels of the I/O stack, primarily MPI and POSIX.

Early approaches to true system self-management relied on the direct classification of system state or behavior to automatically diagnose problems or even enact optimization policies. The work of Madhyastha and Reed [2] compares classifications of I/O access patterns, from which higher level application I/O patterns are inferred and looked up in a table to determine the file system policy to set for the next accesses. The table, however, has to be supplied by an administrator implementing his own heuristics.

Later approaches are marked by schemes to persist their results in order to benefit from past diagnostic efforts, possibly even applying known repair actions to recognized problems. Magpie, a system by Barham et al. [3], traces events under Microsoft Windows, merging them according to predefined schemata specifying event relationships. Their causal chains are reconstructed, attributed to external requests and clustered into models for the various types of workload observed. Deviations will point to anomalies deserving human attention.

Yuan et al. [4] combine system state and system behavior to identify the root causes of recurring problems. Tracing the system calls generated under Windows XP, they use support vector machines to classify the event sequences. A

presumptive root cause is identified, leaving the sequence – if flagged by a human as accurately diagnosed – available as eventual new training case for the classifier. The root cause description may include repair instructions, which in some cases can be applied automatically.

Of the systems focusing on system metrics, *Cluebox* by Sandeep et al. [5] analyses logs for anomalies, pointing out the system counters most likely involved in the problem. Expected latencies can be predicted for new loads, not only detecting anomalies but also which counters most significantly deviate from par. No direct tracing or causal inference are needed, but once again, only hints for administrators are produced. In *Fa* (Duan et al. [6]), a system’s base state is defined by service level objectives; compliance constitutes health, violation failure. A robust data base of failure signatures is constructed from periodically sampled system metrics.

Behzad et al. [7] offer a framework that uses genetic algorithms to auto-tune select parameters of an HDF5/MPI/Lustre stack; but its monolithic view of the system disregards the relations between the layers as well as the users’ individual requirements, setting optimizations once per application run.

Valuable capabilities of existing approaches are combined in the SIOX infrastructure and extended by unique features: 1) SIOX covers parallel I/O on client and server level as well as intermediate levels, 2) it aims to be applicable at all granularities and portable across middle-ware and file systems, 3) SIOX does not require MPI and can be applied to POSIX applications easily 4) it unites user-level and system-level monitoring supporting both views, 5) it applies machine-learning strategies to learn optimizations on-line and off-line and apply them – ultimately without human intervention, 6) SIOX utilizes a system model to estimate performance, 7) it restricts monitoring to relevant anomalies, 8) finally, SIOX is extremely modular and its capabilities can be configured for many different use-cases.

3 The Modular Architecture of SIOX

Under SIOX, a system will collect information on I/O activities at all instrumented levels, as well as relevant hardware information and metrics about node utilization. The high-level architecture of SIOX has been introduced in [8]: SIOX combines on-line monitoring with off-line learning; monitoring data is first transferred into a transaction system, and then imported into a data warehouse for long-term analysis. The recorded information will be analyzed off-line to create and update a knowledge base holding optimized parameter suggestions for common or critical situations. During on-line operations, these parameters may be queried and used as predefined responses whenever such a situation occurs. The choice of responses to every situation is diverse, ranging from intelligent monitoring in the presence of anomalies, via alerting users and administrators with detailed reports, to automatically taking action to extract the best possible performance from the system. In Zimmer et.al. [9] we discussed the knowledge path in more detail, and sketched several modules for anomaly detection. This paper

extends our previous theoretical articles by describing our existing prototype, and presenting first results.

SIOX is written in C++, and heavily relies on the flexible concept of modules: Upon startup of either a process, a component, or the daemon, a configuration is read which describes the modules that must be loaded. Several modules offer an interface for further plug-ins which, for example, may offer specialized detection of anomalies, and may trigger actions based on the observed activities and system state. Before we outline the currently existing modules and plug-ins, the instrumentation of an application is described.

3.1 Low-Level API

An instrumented application is linked against at least one instrumented software layer, and against the low-level C-API. Upon startup of the SIOX low-level library, a configuration file is read, and globally required modules are loaded. Whenever a logical component – such as the MPI layer – registers, a component-specific section in the configuration is read, and the layer-specific modules described there are loaded. The following interfaces are directly required by the low-level library:

- *Ontology*: The ontology module provides access to a persistent representation of activity attributes such as function parameters.
- *SystemInformationGlobalIDManager*: Provides a means to map existing physical hardware (nodes, devices), and software components (layer and existing activity types) to a global ID, and vice versa.
- *AssociationMapper*: While the Ontology and SystemInformation are rarely updated, runtime information of an application changes with each execution. Therefore, this data is held in a separate store with an efficient update mechanism, both provided by the AssociationMapper.
- *ActivityMultiplexer*: Once an activity is completed, it is given to the multiplexer, which forwards it to all connected MultiplexerPlugins. Each of these analyzes the data of the incoming activities to fulfill higher-level duties. The multiplexer offers both, a synchronous and an asynchronous data path. The latter allows for more performance-intense analysis, but may incur loss of activities when the system is overloaded.
- *Optimizer*: The optimizer provides a lightweight interface to query the best known value for a tunable attribute. Internally, a value is provided by a plug-in; each plug-in may support a set of attributes. The value may depend on the system status, runtime information about the process, or sequence of activities observed.
- *Reporter*: Modules can collect some information about their operation, such as overhead, and amount of processed data. Upon termination of an application, this data is handed over to so-called reporter modules. A reporter may process, store, or output these statistics, according to its configuration.

Additionally, the library uses some helper classes to build activities, to load and manage modules, and to monitor the internal overhead.

3.2 Instrumentation

When tracing foreign code, keeping the instrumentation up to date can become a maintenance problem. To keep manual work to a minimum, SIOX automatically generates instrumentations for layers from annotated library headers. In the modified headers, short annotations represent aspects of otherwise complex code. It is also possible to inject code, or include custom header files. An example of these annotations is given in Listing 1. Each annotation set before a function signature instantiates a template with the given name and defines its arguments.

The `siox-wrapper-generator`, a dedicated python tool, either creates code suitable for ld's `--wrap` flag or for use with `LD_PRELOAD`. The tool uses so-called templates to turn annotations into source code, thus a single annotated header allows for many different outputs by switching templates.

```
//@activity
//@activity_link_size fh
//@activity_attribute filePosition offset
//@splice_before ''int intSize; MPI_Type_size(datatype, &intSize);
//          uint64_t_size=(uint64_t)intSize*(uint64_t)count;,
//@activity_attribute bytesToWrite size
//@error ''ret!=MPI_SUCCESS'' ret
int MPI_File_write_at(MPI_File fh, MPI_Offset offset, void * buf, int count,
                      MPI_Datatype datatype, MPI_Status * status);
```

Fig. 1. Annotations for `MPI_File_write_at()` in the header

We constantly update the capabilities of the instrumentation. At the moment, the instrumentation covers a number of functions in different I/O interfaces: 74 in POSIX, 54 in MPI, 5 in NetCDF and 18 in HDF5. Instrumentation of open and close functions in NetCDF and HDF5 allows SIOX to relate lower-level I/O to these calls. Asynchronous calls are supported by linking the completion of an operation to its start, but require creation of this link in the instrumented interface. A restriction, in this respect, is that we expect that start and end calls are executed by the same thread, but this restriction will be lifted in the future. Concepts to relate calls across process boundaries, e.g. between I/O client and servers, have been developed but are not being used so far.

3.3 Existing Modules

For the basic modules needed by the low-level API, database and file system back-ends are available. The *ActivityFileWriter* and *StatisticsWriter* store the respective information into a private file, persisting observed activities and statistics without the need to set up a database. There are three alternative *ActivityFileWriter* modules with alternative representations available: A text file using the Boost library, our own binary format and a Berkeley DB implementation. These modules can be embedded in the daemon to all activities of multiple processes in a file or in the process to store its activities independent of other processes. With the *DatabaseTopology*, we have implemented a PostgreSQL database

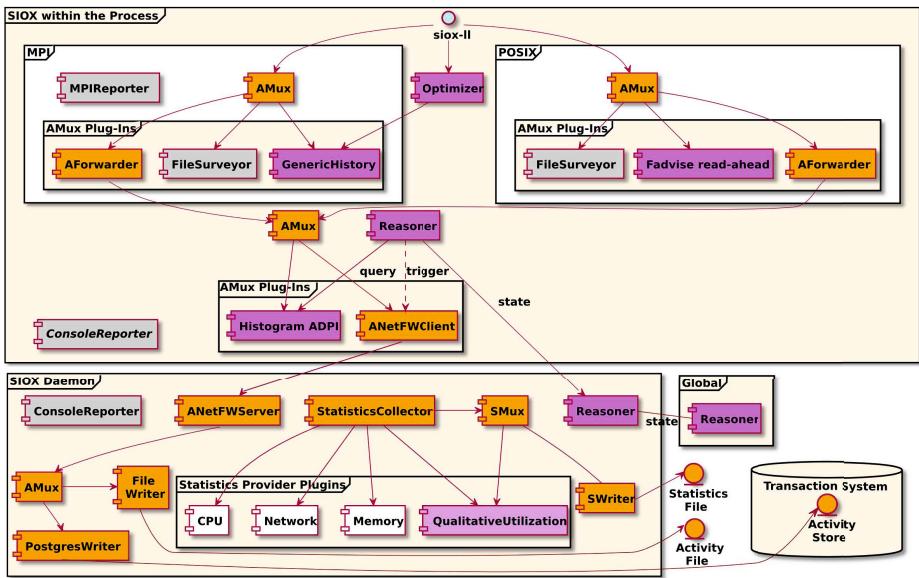


Fig. 2. Example configuration of SIOX modules within a process and the node-local daemon, and their interactions. Those used for monitoring are styled orange or white, those for self-optimization purple; utility modules are gray.

driver for key-value like tables. This module offers a convenient interface for a *TopologyOntology*, *TopologySystemInformation*, and *TopologyAssociationMapper*. Also, the *PostgresWriter* stores the activities into a PostgreSQL database.

The *GIOCommunication* module handles all communication within SIOX. It uses GLIB IO sockets, thus offering both TCP/IP and Unix sockets connections. For each communication partner, two threads are started: one handling incoming messages, and one for transmission.

The typical configuration and interactions of higher-level modules are illustrated in Figure 2. In this setup, a client with POSIX and MPI instrumentation transfers observations to a node-local daemon, which, in turn, injects the activities into the transaction system. Additionally, system statistics are gathered by the daemon. The responsibility of these modules is briefly described in the following:

- *Optimizer*: The current optimizer implementation is very lightweight, dispatching the requests for optimal parameters to plug-ins.
- *ActivityMultiplexerAsync*: This implementation of an activity multiplexer provides both a synchronous processing of completed activities and spawns a thread for background processing of asynchronous notification of registered activity plug-ins. Due to potentially concurrent execution of activities, each plug-in is responsible for protecting critical regions.

- *Reasoner*: Any reasoner will play one of three roles, indicating its scope of responsibility: Process, node, or system (global) reasoner. They periodically poll each *AnomalyDetectionPlugIn* within their respective domain for an aggregated view of all anomalies witnessed during the last polling cycle. These will be related to the latest generalized health reports of neighboring reasoners in the SIOX hierarchy to form a comprehensive view of the local subsystem’s health. In this context, the collected anomalies are evaluated, and the decision is made whether to signal an anomaly to all registered listeners. Although the current standard implementation has a very simple decision matrix with only a few heuristic rules, later versions will play a crucial part in regulating the stream of log data.
- The *ANetFWClient* provides a ring buffer to store a number of recently observed activities. A connected reasoner may emit an anomaly signal which causes the *ANetFWClient* to transmit all pending activities to a remote *ANetFWServer*. Several configurations are possible; at the moment, the process only sends its data to a daemon if the process-internal reasoner decides to do so. In another configuration, a process may always send its data to the node-local daemon which forwards it if an anomaly is detected at node level.
- *GenericHistory*: A plug-in monitoring accesses and the hint set which was active during their execution. After a learning phase, it can identify the hint set most advantageous to a given operation’s performance in the past; these can be further conditioned on user ID and file name extension. The optimizer will query this plug-in, and inject commands to set the hints appropriately before each access; this requires instrumenting the layer issuing the access calls for SIOX. The calls to be observed, and the attributes governing their performance (such as data volume or offset) are configurable.
- *Histogram ADPI*: This plug-in either learns a typical runtime histogram for each type of activity, or it reads the required data from the database. This data is then used to categorize the speed for subsequent activities into very slow, slow, normal, fast, and very fast operations. An aggregate view of this information is supplied to the reasoner, which may then judge the overall system state in turn.
- *FileSurveyor*: Activities of the classes *Read*, *Write*, and *Seek* are monitored here, counting sequential (further distinguished by stride size) and random accesses, and reporting the totals upon application termination. The calls belonging to each class may be configured according to the layer surveyed.
- *FadviseReadAhead*: This plug-in tracks POSIX I/O, and may decide to inject `posix_fadvise()` calls to read-ahead future data. The decision is made based on the amount of data accessed in the previous call, and the spatial access pattern. For each file, it predicts the next access position of the stream, and initiates a call to `fadvise()` if a configurable number of preceding predictions have been correct.
- Statistics infrastructure: Statistics are provided by *StatisticsProviderPlugins*. Their task is to acquire the data, tag it with ontology attributes and topology paths, and to make it accessible to SIOX via a simple interface. A *StatisticsCollector* then polls all the providers registered with it; the current

implementation uses a dedicated thread for this purpose. Afterwards, the collector calls a *StatisticsMultiplexer* to distribute the statistics information to its listeners.

All *Statistics* can remember values from the near past, going back as far as 100 minutes. As our polling interval for statistics is 100 milliseconds, we cannot store the entire history at full resolution. Consequently, the statistics data is aggregated into longer intervals, providing five different sampling frequencies, each storing its last ten samples, and each serving as the basis to aggregate at the next level. With this approach, it is possible to ensure a reasonable memory overhead while providing long history information. The sampling intervals used in SIOX are 100 milliseconds, 1 second, 10 seconds, 1 minute, and 10 minutes.

- *StatisticsProviderPlugins*: Currently, five StatisticsProviderPlugins are available, collecting information on CPU, memory, network, and I/O load. The fifth plug-in is the QualitativeUtilization plug-in, which acts both as a StatisticsMultiplexerListener and a StatisticsProviderPlugin, integrating the detailed information supplied by the specialized plug-ins into four simple high-level percentages describing the relative utilization of CPU, network, I/O-subsystem and memory.

4 Analysis and Visualization of I/O

Post-mortem and near-line analysis of observed activity in files and the database are crucial. At the moment, we offer a command-line trace reader and a database GUI. Additionally, each SIOX-instrumented application and the daemon gather statistics about their own usage and overhead. This reporting data is usually output during termination of a process.

4.1 Command-Line Trace Reader

The command-line trace reader offers a plug-in interface to process monitored activities. The *print* plug-in just outputs all recorded trace information, replacing attributes with their human-readable representations. An excerpt of a trace is given in Figure 4; note the cause and all potential relations of an activity printed at the end of each line. Another plug-in, the *AccessInfoPlotter* extracts the observed spatial and temporal access pattern for each process and uses pyplot to illustrate the I/O behavior. Figure 3 shows the interleaved I/O of two processes each writing 20×100 KiB blocks with one non-contiguous MPI_File_write(). Thanks to data sieving, these patterns lead to several read-modify-write cycles (500 KiB of data per iteration). Both traces have been channeled through a local daemon, and are stored in a single trace file.

4.2 Database GUI

The visualization of the large amount of data produced by a fully functional deployment of SIOX is a challenging and resource-intensive task that exceeds

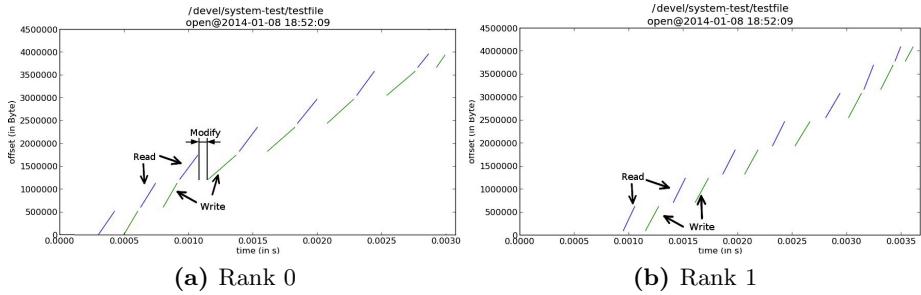


Fig. 3. Generated plots for the observed POSIX access pattern from a shared file accessed by two processes using non-contiguous I/O. Each process locks a file region, reads the data, modifies it and writes it back. Selected phases are marked with arrows to illustrate the behavior.

```
0.0006299 ID1 POSIX open(POSIX/descriptor/filename="f1",POSIX/descriptor/filehandle=4) = 0
0.0036336 ID2 POSIX write(POSIX/quantity/BytesToWrite=10240, POSIX/quantity/BytesWritten=
10240, POSIX/descriptor/filehandle=4, POSIX/file/position=10229760) = 0 ID1
0.0283800 ID3 POSIX close(POSIX/descriptor/filehandle=4) = 0 ID1
```

Fig. 4. Example trace output created by the trace-reader. ID* is the locally generated ID (shortened in this example). The relation between `open()` and the other calls is recorded explicitly.

the scope of the project. However, the possibility of a user-friendly inspection of the data stored in the database is essential for the development and administration of the SIOX system. For this reason, we created a web interface based on HTML/PHP that extracts and presents the information we are interested in. Giving the simplicity of its implementation, this interface is extensible without much programming effort, resulting in a useful tool for experimenting with the collected data as well as for debugging the system. Currently, the interface offers a listing of all activities stored in the database (see Figure 5), as well as a detailed view of any particular activity together with the causal chain of sub-activities it produced (Figure 6).

4.3 Reporting

By way of the Reporter module, any SIOX component may compile a report upon component shutdown. The *ConsoleReporter* module will collect all reports, and write them to the console for later inspection by the user. Report data is structured into groups, and every field can be accessed separately, allowing for further processing; the *MPIReporter* module, for instance, aggregates reports over several nodes, computing a minimum, maximum, and average for every numeric value reported. Figure 7 demonstrates some statistics collected by the FileSurveyor plug-in for a single file during a Parabench [10] run.

Activity Overview

Purge database | Execution Overview | Time frame statistics

#	Function	Time start	Time stop	Duration [μs]	Error code
19691	MP_Init	27.03.2014 17:47:16 936222147	27.03.2014 17:47:17 287118274	350896.127	
19690	fopen	27.03.2014 17:47:16 937067853	27.03.2014 17:47:16 937353100	285.247	
19689	fileno	27.03.2014 17:47:16 937370065	27.03.2014 17:47:16 937370688	0.623	
19692	fileno	27.03.2014 17:47:16 940894904	27.03.2014 17:47:16 940895669	0.765	
19693	fread	27.03.2014 17:47:16 940989834	27.03.2014 17:47:16 941027243	37.409	
19694	fread	27.03.2014 17:47:16 942210703	27.03.2014 17:47:16 942214476	3.773	
19695	fileno	27.03.2014 17:47:16 942290985	27.03.2014 17:47:16 942291588	0.603	
19696	fileno	27.03.2014 17:47:16 942366812	27.03.2014 17:47:16 942367420	0.608	
19697	fclose	27.03.2014 17:47:16 942418918	27.03.2014 17:47:16 942461562	42.644	
19699	mmap	27.03.2014 17:47:16 949855800	27.03.2014 17:47:16 949881326	25.526	
19701	fopen	27.03.2014 17:47:16 951151207	27.03.2014 17:47:16 951159795	8.588	
19700	fileno	27.03.2014 17:47:16 951163967	27.03.2014 17:47:16 951164515	0.548	
19702	fgets	27.03.2014 17:47:16 951292320	27.03.2014 17:47:16 951344414	52.094	

[first → previous](#) [next → last](#)

Fig. 5. Activity list showing I/O function and timestamps

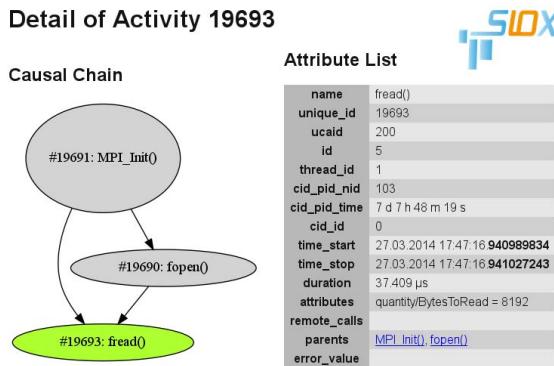


Fig. 6. Detailed view of activity showing the causal chain and list of attributes

5 Experiments

In this section, we analyze the overhead of the SIOX infrastructure, and discuss two use-cases in which SIOX already improves performance.

5.1 System Configuration

The experimental configuration on the WR cluster consists of 10 compute nodes (2×Intel Xeon X5650@2.67GHz, 12 GByte RAM, Seagate Barracuda 7200.12) and 10 I/O nodes (Intel Xeon E3-1275@3.40GHz, 16 GByte RAM, Western Digital Caviar Green WD20EARS) hosting a Lustre file system. A dual-socket compute node provides a total of 12 physical cores and 24 SMT cores. All nodes are interconnected with gigabit ethernet, thus, the maximum network throughput between the compute and the I/O partition is 10 × 117 MB/s. As a software basis, we use Ubuntu 12.04, GCC 4.7.2, and OpenMPI 1.6.5 with ROMIO. For the overhead measurement, we compiled SIOX using -O2.

```
[FileSurveyor:15:"MPI Generic"] "/mnt/lustre//file.dat"/Accesses = (40964,40964,40964)
[FileSurveyor:15:"MPI Generic"] "/mnt/lustre//file.dat"/Accesses/Reading/Random, long seek = (20481.8,20480,20482)
[FileSurveyor:15:"MPI Generic"] "/mnt/lustre//file.dat"/Accesses/Reading/Random, short seek = (0,0,0)
[FileSurveyor:15:"MPI Generic"] "/mnt/lustre//file.dat"/Accesses/Reading/Sequential = (0.2,0,0.2)
[FileSurveyor:15:"MPI Generic"] "/mnt/lustre//file.dat"/Bytes = (8.38861e+09,8.38861e+09,8.38861e+09)
[FileSurveyor:15:"MPI Generic"] "/mnt/lustre//file.dat"/Bytes/Read per access = (204780,204780,204780)
[FileSurveyor:15:"MPI Generic"] "/mnt/lustre//file.dat"/Bytes/Total read = (4.1943e+09,4.1943e+09,4.1943e+09)
[FileSurveyor:15:"MPI Generic"] "/mnt/lustre//file.dat"/Seek Distance/Average writing = (1.0238e+06,1.0238e+06,1.0238e+06)
[FileSurveyor:15:"MPI Generic"] "/mnt/lustre//file.dat"/Time/Total for opening = (3.9504e+08,3.66264e+08,4.38875e+08)
[FileSurveyor:15:"MPI Generic"] "/mnt/lustre//file.dat"/Time/Total for reading = (1.47169e+11,1.0968e+11,1.7617e+11)
[FileSurveyor:15:"MPI Generic"] "/mnt/lustre//file.dat"/Time/Total for writing = (1.08783e+12,1.03317e+12,1.16192e+12)
[FileSurveyor:15:"MPI Generic"] "/mnt/lustre//file.dat"/Time/Total for closing = (1.0856e+11,6.11782e+10,1.46834e+11)
[FileSurveyor:15:"MPI Generic"] "/mnt/lustre//file.dat"/Time/Total surveyed = (1.34568e+12,1.34568e+12,1.3457e+12)
```

Fig. 7. Example report created by FileSurveyor and aggregated by MPIReporter (shortened excerpt). The number format is (average, minimum, maximum).

5.2 Instrumentation of Concurrent Threads

The software instrumentation of SIOX adds overhead to the critical path of applications. To assess this overhead, we created a multi-threaded benchmark: each thread calls `fwrite()` without actually writing any data. Without instrumentation, a single call needs roughly 6 ns or 14 CPU cycles. The overhead of the instrumented `fwrite()` is visualized in Figure 8 for 1 to 24 threads and three different configurations. *SIOX plain* refers to an almost empty configuration without additional modules, in *SIOX POSIX fw* we add a configuration section for POSIX containing the AForwarder and the additional AMux for POSIX. In the *SIOX process* configuration, we enabled all the modules inside a process as shown in Figure 2 (the Reasoner is not included here because it contains only a low-frequency periodic thread).

Since we must protect critical regions for each module, an increasing number of threads compete for these resources, leading to contention. In our measurements, the critical section accounts for a runtime of about $0.75\ \mu\text{s}$ (plain configuration) to about $6\ \mu\text{s}$ (process configuration). As every activity is intended to be transferred to the node daemon, if an anomaly occurs, this path may impose a bottleneck. A benchmark of the communication module demonstrates a sustained transfer rate of 90,000 messages/s (1 KiB payload). Thereby, it should handle typical I/O scenarios. The file writer modules store activities with a rate of 70,000 activities/s, thus they could persist the activities on node level.

We did not include the database back-end for activities in our measurements because our PostgreSQL instance on a VM just inserts about 3,000 activities/s and thus is a bottleneck we are working on. Regardless of the low integration of activities into our transaction system; as these processes are usually done by the daemon, they happen concurrent to the application execution not deferring application I/O.

5.3 Instrumentation of the ICON Climate Model

In addition to our benchmark experiments, we also measured the impact of SIOX on the runtime of a climate model. For this test, we used the ICON model [11] developed by the Max Planck Institute for Meteorology (MPI-M) and the German Weather Service (DWD). The test setup was as follows: First

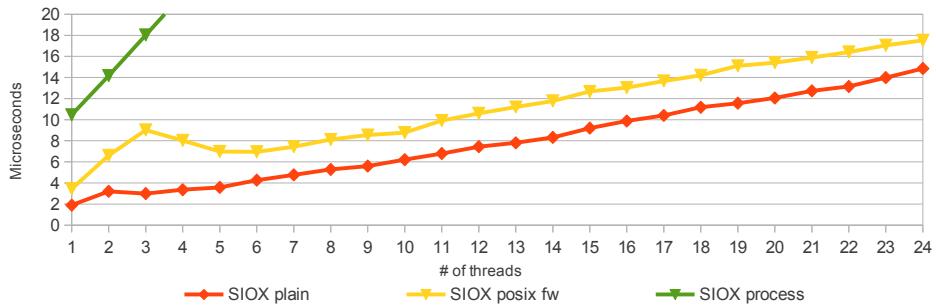


Fig. 8. Overhead per thread due to critical regions in the modules

we simulated one day using a 20480 cell icosahedral grid (ICON’s R2B04 grid), utilizing all 12 physical cores available on one cluster node. This takes 100.7 seconds on average. Then the model was rerun with different levels of SIOX instrumentation using LD_PRELOAD. This was repeated ten times to get runtime measurements that were precise enough to be interpreted.

The resulting times show an overhead between 2.5 and 3.0 seconds when only POSIX or MPI were instrumented and 5.0 seconds for both. Measurement errors ranged from 0.29 seconds to 0.65 seconds. To assess the amount of relative and absolute overhead, the entire test was repeated with twice the simulation time which takes 193.3 seconds on average without instrumentation. In this test, only the overhead of the pure MPI instrumentation increased to 4.5 seconds, the other two instrumentation overheads increased slightly but remained within their respective error intervals.

Much of the constant overhead is due to the reporting output produced by SIOX or by the initialization of the database connection. The incremental costs of SIOX, however, are barely measurable.

5.4 Instrumentation/Optimization of Parabench: 4 Levels of Access

In this experiment, we instrument the parallel I/O benchmark Parabench with SIOX. Additionally, we sketch a scenario in which SIOX will improve performance by automatically setting MPI hints.

In our strided access pattern, each process accesses 10240 blocks with a size of 100 KiB, which accumulate in a shared file of 10 GiByte. We measure the performance of the four levels of access in MPI. According to [12], they are defined by the two orthogonal aspects: collective vs. independent I/O, and contiguous vs. non-contiguous I/O. The observed performance is illustrated in Figure 9, the four configurations are as follows: First, there is a configuration without user-supplied hints, the second configuration adds hints but no SIOX instrumentation, the third adds MPI instrumentation using SIOX’s 1d wrap option, and the last uses LD_PRELOAD to instrument both, MPI and POSIX.

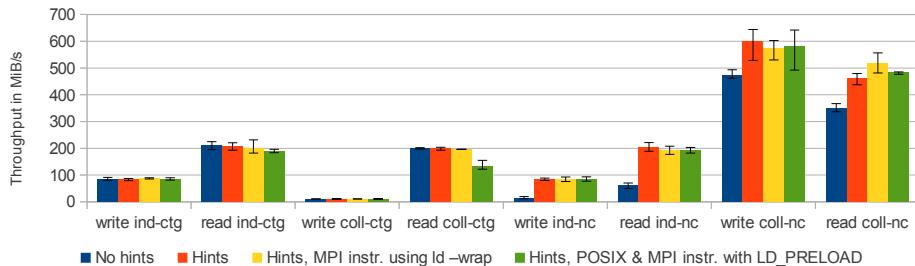


Fig. 9. Performance comparison of the 4-levels of access on our Lustre file system. The configuration with hints increases the collective buffer size to 200 MB and disables data sieving.

On our system, we observe that data-sieving decreased performance significantly (look at the ind-nc cases), therefore we disable it in our hint set.

Overall, the observable performance of the instrumented Parabench is comparable to a regular execution. There is one exception; at 130 MiB/s, the read coll-ctg is behind the normally measured performance of 200 MiB/s. Thanks to our reporting, we could understand the cause. Since we intercept all POSIX I/O operations, the socket I/O caused by MPI is also monitored, adding overhead to each communication¹. The MPI communication causes 650,000 activities per process and is most intense for collective contiguous I/O. In total, I/O accounts to just about 60,000 and 41,000 activities, for POSIX and MPI, respectively. Also, the FadviseReadAhead applies read-ahead hints to socket I/O (without this plug-in, performance improves to 170 MiB/s).

A potential gain for users will be the automatic learning of SIOX, which we just started to explore with the first plug-ins. By virtue of the GenericHistory plug-in, SIOX can already automatically set hints during `MPI_File_open` that have proven to be beneficial in the past. If we modify our benchmark slightly to repeat the test with different hint sets (for example, the default hints, and the improved ones), then the plug-in will remember the improved hints, and set all hints for subsequent opens that do not define them. Thus, without any modification or recompilation of the application, users would benefit from globally known hints.

5.5 Read-Ahead with fadvise

With the FadviseReadAhead plug-in, a module has been implemented which detects a strided read access pattern and injects `posix_fadvise()` to fetch data for the next access – if the last 4 predictions have been correct. To assess its performance, a small benchmark is created which loops over 10 GiB of data stored on a

¹ With the newest version, it is possible to report these operations to a “POSIX_Network” component, handling them differently to I/O operations. However, this is not done in this benchmark.

Table 1. Time needed to read one 1 KiB data block in a strided access pattern

Experiment	20 KiB stride	1000 KiB stride
Regular execution	97.1 μ s	7855.7 μ s
Embedded fadvise	38.7 μ s	45.1 μ s
SIOX fadvise read-ahead	52.1 μ s	95.4 μ s

compute node’s local disk. On each iteration, the benchmark simulates compute time by sleeping a while, then it seeks several KiB forward and reads a 1 KiB chunk – the whole area covered by the seek and one access is defined as the stride size. Two different strides are evaluated: 20 KiB and 1000 KiB. Sleep time is adjusted from 100 μ s in the 20 KiB case to 10 ms to make read-ahead possible. The benchmark is executed several times, between each run the page cache is cleared using `echo 3 > /proc/sys/vm/drop_caches`; the deviation between runs is below 1% of runtime.

In order to validate the results, the `posix_fadvise()` calls issued by the SIOX module have also been embedded into the original source code, thus yielding best performance without any overhead from SIOX. Table 1 compares regular, uncached execution with the manual source code modifications and the FadviseReadAhead module. It can be observed that fadvise improves performance already for 20 KiB strides but excels at 1000 KiB stride, decreasing time per I/O from 7.8 ms to 45 μ s. This improvement can be explained by the data placement: Since EXT4 tries to place logical file offsets close together on the drive’s logical block addressing, the larger stride forces movement of the disk’s actuator. Consequently, with appropriate hints, the programmer can reduce I/O time drastically, but this requires manual adaption of the code. The module shipped with SIOX adds some overhead but improves performance similarly and automatically.

6 Future Work

Opportunities for future work abound. With our basic infrastructure, we can now quickly develop new modules dedicated to certain purposes. Besides new optimizations, we are working on improving the performance of our transaction system which, in turn, we will use to record a large training set of I/O from various benchmarks and applications. Once available, we can improve the rules of our reasoner module, and evaluate machine learning techniques to extract further knowledge. Also, new services will be located in the daemon to cache the necessary information of the global services, such as the ontology.

In the consortium, we are currently working on a first prototype for a GPFS plug-in in OpenMPI which is explicitly instrumented for SIOX. The implementation will also monitor GPFS using the Data Management API framework[13] that is used to keep track of I/O events for selected files and file regions. The SIOX high-level I/O library is designed to not only monitor I/O operations unidirectionally, it is also capable of receiving optimization hints from SIOX at

runtime. Finally, we aim to automatically select the best GPFS hints based on the utilization and historical I/O records from our knowledge bases. However, the various modules imaginable for pattern creation and matching, for anomaly detection and for optimization, as well as the machine learning algorithms, offer a rich field for researchers and system administrators alike.

7 Summary and Conclusions

Our vision for SIOX is a system that will collect and analyze activity patterns and performance metrics in order to assess and optimize system performance. In this paper, we presented results of our prototype and have given a glimpse of its flexible and modular architecture. Although the monitoring with SIOX imposes some overhead in the critical path of I/O, we were able to demonstrate that this overhead is very small compared to the performance gains that may be achieved. The I/O stack offers a large variety of potential optimizations which need to be controlled intelligently, and the SIOX system provides the architecture to do so. In this sense, SIOX is the swiss army knife for experimenting with alternative and automatic I/O optimizations without even modifying existing code. Since the overhead is bearable, we believe that (with appropriate modules and configuration) constant supervision with SIOX will greatly improve performance in data centers.

References

1. Carns, P.H., Latham, R., Ross, R.B., Iskra, K., Lang, S., Riley, K.: 24/7 Characterization of Petascale I/O Workloads. In: Proceedings of the First Workshop on Interfaces and Abstractions for Scientific Data Storage, New Orleans, LA, USA (September 2009)
2. Madhyastha, T., Reed, D.: Learning to Classify Parallel Input/Output Access Patterns. *Parallel and Distributed Systems, IEEE Transactions on* 13(8), 802–813 (2002)
3. Barham, P., Donnelly, A., Isaacs, R., Mortier, R.: Using Magpie for Request Extraction and Workload Modelling. In: Proceedings of the 6th Symposium on Operating Systems Design and Implementation, vol. 6, pp. 259–272 (2004)
4. Yuan, C., Lao, N., Wen, J.R., Li, J., Zhang, Z., Wang, Y.M., Ma, W.Y.: Automated Known Problem Diagnosis with Event Traces. In: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems, EuroSys 2006, pp. 375–388. ACM, New York (2006)
5. Sandeep, S.R., Swapna, M., Niranjan, T., Susarla, S., Nandi, S.: CLUEBOX: a Performance Log Analyzer for Automated Troubleshooting. In: Proceedings of the First USENIX Conference on Analysis of system logs, WASL 2008, USENIX Association, Berkeley (2008)
6. Duan, S., Babu, S., Munagala, K.: Fa: A System for Automating Failure Diagnosis. In: Data Engineering. In: IEEE 25th International Conference on ICDE 2009, March 29-April 2, pp. 1012–1023 (2009)

7. Behzad, B., Huchette, J., Luu, H.V.T., Aydt, R., Byna, S., Yao, Y., Koziol, Q.: Prabhat: A framework for auto-tuning hdf5 applications. In: Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing, HPDC 2013, pp. 127–128. ACM, New York (2013)
8. Wiedemann, M.C., Kunkel, J.M., Zimmer, M., Ludwig, T., Resch, M., Bönisch, T., Wang, X., Chut, A., Aguilera, A., Nagel, W.E., Kluge, M., Mickler, H.: Towards I/O Analysis of HPC Systems and a Generic Architecture to Collect Access Patterns. Computer Science - Research and Development 1, 1–11 (2012)
9. Zimmer, M., Kunkel, J.M., Ludwig, T.: Towards self-optimization in HPC I/O. In: Kunkel, J.M., Ludwig, T., Meuer, H.W. (eds.) ISC 2013. LNCS, vol. 7905, pp. 422–434. Springer, Heidelberg (2013)
10. Mordvinova, O., Runz, D., Kunkel, J., Ludwig, T.: I/O Performance Evaluation with Parabench – Programmable I/O Benchmark. Procedia Computer Science, 2119–2128 (2010)
11. Max-Planck-Institut für Meteorologie: ICON,
<http://www.mpimet.mpg.de/en/science/models/icon.html>
12. Thakur, R., Gropp, W., Lusk, E.: Optimizing Noncontiguous Accesses in MPI/IO. Parallel Computing 28(1), 83–105 (2002)
13. IBM: Data Management API Guide (2013)

Framework and Modular Infrastructure for Automation of Architectural Adaptation and Performance Optimization for HPC Systems

Leonardo Fialho^{1,2,4} and James Browne^{3,4}

¹ Institute for Computational Engineering and Sciences

² Texas Advanced Computing Center

³ Department of Computer Science

⁴ The University of Texas at Austin, TX 78712, USA

fialho@utexas.edu, browne@cs.utexas.edu

Abstract. High performance systems have complex, diverse and rapidly evolving architectures. The span of applications, workloads, and resource use patterns is rapidly diversifying. Adapting applications for efficient execution on this spectrum of execution environments is effort intensive. There are many performance optimization tools which implement some or several aspects of the full performance optimization task but almost none are comprehensive across architectures, environments, applications, and workloads. This paper presents, illustrates, and applies a modular infrastructure which enables composition of multiple open-source tools and analyses into a set of workflows implementing comprehensive end-to-end optimization of a diverse spectrum of HPC applications on multiple architectures and for multiple resource types and parallel environments. It gives results from an implementation on the Stampede HPC system at the Texas Advanced Computing Center where a user can submit an application for optimization using only a single command line and get back an at least, partially optimized program without manual program modification for two different chips. Currently, only a subset of the possible optimizations is completely automated but this subset is rapidly growing. Case studies of applications of the workflow are presented. The implementations currently available for download as the PerfExpert tool version 4.0 supports both Sandy Bridge and Intel Phi chips.

1 Introduction

The execution environments provided by current multicore chips and heterogeneous compute nodes are sufficiently complex that even the best modern compilers, using only the information content in the static source code, cannot always generate efficient code for a wide spectrum of application codes across this diverse spectrum of execution environments. Because of the diversity of execution environments, performance optimization is becoming architectural adaptation.

Successful use of most of the existing performance optimization tools requires their users to have in-depth knowledge of computer architecture, compilers, and

performance optimization and to manually execute the tasks of: diagnosis of the causes of performance bottlenecks, specification of optimizations, and implementation of the optimizations. As a result, users seldom optimize their applications to obtain maximum benefit from the rapidly changing architectures of high performance parallel computers. Many high performance parallel computers incorporate co-processors or accelerators in their compute nodes. The co-processors/accelerators typically have different models of programming or require use of a different programming language and may require use of different tool for optimization. Adaptation of applications for effective use of these co-processors is a laborious and difficult task for most users and application developers while use of these accelerators may provide substantial performance gains. Much past work has focused on chip and node levels of execution environments. However, optimization of communication and I/O may also become increasing important. Optimization of power efficiency is becoming of increasing interest. Therefore, there is a critical need for tools and systems which ease the task of performance optimization and architectural adaptation across a wide spectrum of architectures, execution environments and resource types.

1.1 Approach and Contributions

Performance optimization based on runtime measurement can be implemented by a cyclic workflow consisting of four phases: 1) measurement, 2) analysis and diagnosis, 3) formulation and recommendation of optimizations, and 4) actual implementation of the recommended optimizations. Performance optimization can take place at and across multiple levels of the architecture of an execution environment. There are many open-source tools which implement one or more of these phases. There are however, no previous open-source systems which fully automate all of the phases of performance optimization. This paper presents a system which does fully automate performance optimization for an important subset of execution environments and performance bottlenecks. The infrastructure described in this paper goes beyond automating performance optimization for a single execution environment and enables construction of workflows which can be customized for automated optimization and adaptation for multiple execution environments and resource types.

The approach is to integrate powerful open-source or commercial tools which execute one or more phases of performance optimization for different execution environments into an extendable infrastructure where each logical step in the workflow of the infrastructure can be implemented using existing tools. This approach based on a modular infrastructure was inspired by the need to extend the partial automation implemented by earlier versions of PerfExpert [4] to apply the recommended optimizations and then to meet the need to extend PerfExpert-like capabilities to additional parallel environments such as MPI and OpenMP. The infrastructure arising in this approach can also be viewed as an integration of traditional measurement based performance optimization and guided auto-tuning. Additional discussion of this topic can be found in Section 5 along with

some widely used open-source tools which are available, including those used in the current implementation of the infrastructure.

Automation of performance optimization and architectural adaptation based on runtime measurements requires all four steps of runtime metric based performance optimization automated and incorporated into a workflow. Implementing the workflow requires implementation of each of these tasks, definition of interfaces coupling the outputs and inputs of the tasks, selection of a vehicle for storing and saving the permanent state associated with the workflow and integration of these implementations into a cohesive workflow. The knowledge base underlying automation of these steps include:

1. Specification of metrics which combined with knowledge of the structure of the source code and the execution environment enable:
 - (a) Systematic identification of and diagnosis of the causes of performance bottlenecks in localized code segments.
 - (b) Formulation and specification of optimizations/adaptations local to specific code segments/patterns.
 - (c) Implementation of architecturally specific, environment specific, and code structure specific optimizations.
2. Specification of the measurements necessary to evaluate these metrics.
3. A set of rules (parameterized by metrics and based on a model of the architecture and code segment) for diagnosis of the cause of the performance bottleneck and formulation and ranking of possible optimizations based upon metrics and the pattern of the source code associated with each bottleneck.

Formulation of this knowledge base requires knowledge of computer architecture, compilers, artificial intelligence, parallel programming, and performance optimization. Formulation of this knowledge base for a given execution environment enables implementation of an instance of the workflow customized to this environment provided the tools for implementing each phase of the workflow are available. The fundamental contributions reported in this paper include:

1. Specification of a systematic end-to-end workflow and modular infrastructure for performance optimization of applications for multiple execution environments (including co-processors and accelerators) and resource types.
2. A completely open-source and extendable implementation of this workflow.
3. Methods for complete specification implementation, and integration into the workflow of a common subset of source-to-source optimizing transformations.

The remainder of this paper is organized as follows. Section 2 presents the framework, how its modular infrastructure can be extended, and the general workflow operation. Section 3 details the base infrastructure and capabilities designed and implemented to automatic performance optimization. It also describes how the capabilities for automatic performance optimization may be extended. Section 4 shows case studies where the capabilities of the framework and its modular infrastructure have been successfully applied. Section 5 defines and describes related research. Section 6 sketches future research on and functionalities in the infrastructure. Section 7 summarizes the contributions and lessons learned.

2 Framework, Modular Infrastructure, and Workflow

The workflow is implemented with a functionally partitioned design. Each step in the optimization process is separate with an interface defined by metadata. There are 5 steps in the workflow. The first step is the initialization of the workflow which may include compilation of the application. The next four steps in the workflow (measurement, analysis, optimization strategy, and optimization implementation) actually implement the optimization process.

The inputs and outputs of each step in the workflow are stored in a relational database according to the metadata defining the inputs and outputs of each step. This structure enables not only updates and enhancements of each optimization step independently of the others, but it also enables replacement of the tools implementing each step to customize the workflow to a specific execution environment. For example, replacement of an architecture-specific performance measurement tool by another measurement tool for a different architecture.

The modular infrastructure design also makes it easy to extend the capabilities of the workflow with additional recommendations for optimization, rules to select such recommendations, as well as tools to actually implement them either by source code transformation or by tuning the parallel environment.

The framework and its workflow, which automates performance optimization, are schematically represented in Figure 1. Execution of the workflow begins with a single command line with parameters which specifies the options for execution of the workflow the user has chosen.

The initial step initializes the workflow which by default parameter will compile the application if the source code is provided. The application is then invoked under the control of a measurement module to collect runtime measurements.

The measurement and analysis steps in the workflow are specified with a modular structure which encapsulates measurement tools and analysis tools and coordinates their interactions through a feedback loop. Each measurement tool must also attribute the measurements it makes to source code constructs. Each time a measurement module is invoked it collects some specified set of measurements (*e.g.*, performance counters, memory access behavior, MPI communication patterns) and maps them to local code segments (*e.g.*, functions, loop nests).

The analysis modules convert the measurements to standard metrics, compare the metrics to norms, identify the local code segments which use significant compute time and resources, and for those code segments which use one or more resources inefficiently, attempt to determine the cause of the performance bottleneck. If the first set of measurements are not sufficient to diagnosis the performance bottlenecks for the code segments which are indicated to have performance bottlenecks, then the analysis modules may re-invoke another measurement module to generate additional measurements. The analysis/diagnosis module will then compute a set of metrics for each execution environment or different type of resource use (*e.g.*, local cycles per instructions – LCPI, cache memory reuse and cache thrashing, memory access stride, thread imbalance).

When the measurements and analyses phases are complete, then, the workflow step formulates, recommends, and ranks optimizations to alleviate the performance bottlenecks in each local code segment is executed. This phase is composed of a set of rule-based strategies which combine the required metrics to provide a ranked list of possible optimizations for each performance bottleneck. The optimization step in the workflow begins with static analysis to verify the safety and the feasibility of applying the recommended optimizations and then initiates a process to implement the top recommendation for optimizing each code segment which is a bottleneck. The optimization process is then re-initiated and terminates when the optimization step cannot apply any of the recommended optimizations or the system exhausts its library of recommendations.

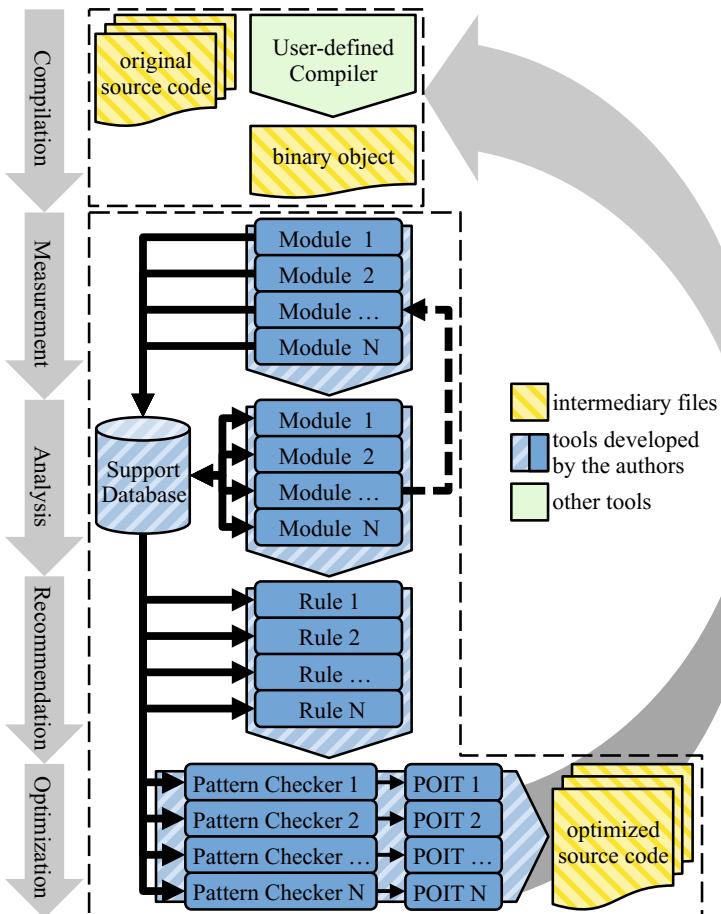


Fig. 1. Workflow for automation of performance optimization showing the four steps required plus compilation. The modular infrastructure is also represented on each of the four conceptual steps of automation of performance optimization.

3 Workflow Realization

To enable multiple implementations for different chip architectures with a reasonable level of effort, it is essential to be able to use existing measurement tools. The analysis tools map the raw measurements into metrics which can be used for diagnosing performance bottlenecks and specifying optimizations for alleviating the bottlenecks. This section describe these first two steps. Section 3.3 describes how a relational database is used to integrate and unify the steps of the workflow. Section 3.4 describes the third step in the workflow which implements rules for selection and ranking of optimizations. Section 3.5 describes the process for implementing the optimizations in the application. Section 3.6 describes the workflow management program which coordinates the execution of the several steps.

3.1 Measurement and Analysis Module Interfaces and Interactions

The measurement and analysis modules, which are dynamic shared libraries, implement three interfaces (`load`, `init`, `fini`) and, at least, one of the: compile, measurements, and analysis interfaces. `load`, `init` and `fini` are interfaces used by the infrastructure to load modules, initialize modules, and finalize modules, respectively. The `measurement` and `analysis` interfaces represent the functionality that the module provides for the optimization process while the `compile` interface is used to include some tool-specific requirement (*e.g.*, a measurement tool which required instrumentation of relinking). Modules which implement these interfaces will be invoked by the infrastructure during the appropriate phase. Note that a single existing tool such as MACPO [23] which gathers measurements through instrumentation of the source code may require multiple interfaces. The interfaces are defined as functions in a C structure which represents the base module. Modules are specific implementations derived from this base module, which may also extended the base module with additional interfaces. Extended interfaces are useful to allow interaction between modules (*i.e.*, one module direct calling an interface of another module). The base module also implements a set of utility functions to help module developers. Examples are to query the list of available modules, retrieve user-defined command line arguments, check the version and status of other modules, etc.

3.2 Currently Implemented Measurement and Analysis Modules

Currently there are five measurement and analysis modules implemented in the infrastructure: 1) `hpctoolkit` [29] which encapsulates HPCToolkit, 2) `vtune` [24] which encapsulates Intel VTune, 3) `lcp` which implements an analysis based on local cycles per instruction [4], 4) `macpo` which encapsulates both MACPO measurements and analysis, and 5) `readelf` which interfaces with a DWARF library.

hpctoolkit: this module implements the **measurement** interface and executes the application with HPCToolKit multiple times to gather all the performance data required to compute the metrics for a given architecture. It outputs two tables (**hpctoolkit_hotspot** and **hpctoolkit_event**) into the database with the application profile (*i.e.*, basically the application function call trace) and the events associated to each hotspot (function or loop).

vtune: this measurement module is very similar to the **hpctoolkit** module, except that it uses Intel VTune to collect hardware performance counters.

lcpi: this module implements both the **measurement** and **analysis** interfaces. The functionality of the **measurement** interface is to determine the set of performance counters which must be gathered for the chips of the execution environment to enable computation of the LCPI metric (partially shown in Table 1) and put them in the database. The functionality of the **analysis** interface uses as input the table of hotspots generated by the **hpctoolkit** module or **vtune** module and a table which contains the chip characterization data (generated during the installation process) in order to compute the LCPI metric. During the analysis phase this module computes and outputs a table (**lcpi_metric**) with the LCPI metrics for all the hotspots from the input table. The computed metrics are also presented to the user as a report, partially depicted in Listing 1.1. It supports both host processors (x86 only) and Intel Xeon Phi coprocessors.

macpo: the MACPO module implements **compile**, **measurement** and **analysis** interfaces. During the compilation phase it instruments the functions and loops designated as performance bottlenecks by the LCPI metric values. While collecting measurements it runs the application with MACPO instrumentation to get the memory access trace. During the analysis phase it generates the memory centric metrics, which are stored in a table into the support database (**macpo_metric**) and used to generate a report to the user.

readelf: this module implements the **measurement** interface which interacts with a DWARF library using the application binary as input to identify the programming language and the compiler used to generate the executable file, which is then stored in a table into the knowledge database (**readelf**).

Table 1. Data access LCPI metrics for Sandy Bridge chips. **L1_dlat**, **L2_lat**, **L3_lat**, and **mem_lat** are architecture-dependent parameters calculated by microbenchmarks.

Name	Model for HPCToolkit module (PAPI event names)
overall	$((\text{PAPI_LD_INS} * \text{L1_dlat}) + (\text{PAPI_L2_DCA} * \text{L2_lat}) + ((\text{PAPI_L3_TCA} - \text{PAPI_L3_TCM}) * \text{L3_lat}) + (\text{PAPI_L3_TCM} * \text{mem_lat})) / \text{PAPI_TOT_INS}$
L1 hits	$(\text{PAPI_LD_INS} * \text{L1_dlat}) / \text{PAPI_TOT_INS}$
L2 hits	$(\text{PAPI_L2_DCA} * \text{L2_lat}) / \text{PAPI_TOT_INS}$
L3 hits	$((\text{PAPI_L3_TCA} - \text{PAPI_L3_TCM}) * \text{L3_lat}) / \text{PAPI_TOT_INS}$
LLC misses	$(\text{PAPI_L3_TCM} * \text{mem_lat}) / \text{PAPI_TOT_INS}$

```

1 Loop in function tdma in tdma.f90:55 (80.93% of the total runtime)
2 =====
3 ...
4 LCPI      (interpretation varies according to the metric)
5 * data accesses   1.19 [>>>>>>>>>>>>>]
6 - L1 cache hits  0.15 [>>> . . . . . . . . . .]
7 - L2 cache hits  0.25 [>>>> . . . . . . . . . .]
8 - L3 cache hits  0.18 [>>>. . . . . . . . . .]
9 - LLC misses     0.61 [>>>>>>>>> . . . . . . . . . .]
10 ...
11 -----

```

Listing 1.1. Fragment of the report produced by the LCPI analysis module

3.3 Knowledge Database

The knowledge database is a relational database which stores the information provided and required by all of the steps in the performance optimization process. Each module is responsible for defining the format in which the information it generates is stored. Metadata defining each format is also stored in the database. During PerfExpert installation, the architectural parameters generated by execution of microbenchmarks for each execution environment are stored into the database. Measurement modules collect raw measurement from the application execution behavior and store them on the database. Analysis modules retrieve the measurements from the database to generate metrics, which are then stored on the database. The recommendation step reads the metrics, generates its recommendations and their rankings and stores them in the database. The database also stores the current set of optimizations available and is the engine to run the rules to select recommendations for optimization. This database is what gives to the infrastructure the ability to incorporate the expert knowledge and practical experience upon which selection and ranking of optimizations is based.

Any relational database can be used to implement the knowledge database. In our implementation we chose SQLite version 3.

3.4 Diagnosis of Performance Bottlenecks and Selection of Recommendations for Optimization

The measurements taken by each measurement module are mapped to a set of metrics which are used by analysis modules for diagnosis of the causes of the performance bottleneck and the optimizations to be recommended for alleviating the bottleneck. The measurements taken and the metrics computed may vary across both measurement tools and chip architectures. Selection of recommendations for optimizations are implemented as a set of rules which are conditional functions over metrics and characteristics of the source code segment, architecture, and programming language from which the metrics are derived. The output of the recommendation rules is a ranked set of recommendations to alleviate the bottlenecks. These recommendations may be a modification on the source code,

suggestions to add compiler flags, or even parameters to tune the parallel runtime system (*e.g.*, OpenMP runtime system variables).

The rules to select and rank recommended optimizations are implemented as SQL queries. One example is shown in Listing 1.2. These queries may use the raw measurements and the computed metrics, previously stored on the database by the appropriate modules, as well as the environment characterization (*e.g.*, chip, memory, network) data generated by performance benchmarks executed during the module installation (stored in the `hound` table into the database). Listing 1.2 shows one of these rules which uses the six categories of resource use defined by the LCPI metric analysis module which selects the category which contributes the most to the poor performance of a code section of a given application [28]. The infrastructure provides a set of source code characterization tokens that can be used in SQL queries (*e.g.*, `@LPD` and `@RID` which are the loop depth and the tuple identification on the metrics table where are stored the source code reference metrics, respectively).

While the aforementioned rule tries to select the best optimization given a set of measurements, it is also possible to define very specific rules, such as: `SELECT 49 AS recommendation_id, 10 AS score FROM macpo_metric AS m, hound AS h WHERE (m.streams > (0.8 * h.streams)) AND (0 < @LPD)`. This rule selects a specific optimization to reduce the number of simultaneous streams within a loop (by splitting the loop into multiple loops) if the number of simultaneous streams identified by the MACPO module is greater than supported simultaneous streams in the current architecture (measured during the environment characterization performed while PerfExpert is installed).

Currently, a basic installation of PerfExpert has 53 recommendations pre-loaded into the database (42 related to code patterns, 4 architecture-related, and 7 compiler-specific), 1 rule to select recommendations which tries to rank all these recommendations (shown in Listing 1.2), and 4 architecture-specific rules which take into consideration environment characterization data (*e.g.* L1 cache size,

```

1 SELECT r.id AS recommendation_id, SUM(
2   (CASE c.short WHEN 'd-11' THEN (m.L1d_hits - (max*0.1)) ELSE 0 END) +
3   (CASE c.short WHEN 'd-12' THEN (m.L2d_hits - (max*0.1)) ELSE 0 END) +
4   (CASE c.short WHEN 'd-mem' THEN (m.L2d_misses - (max*0.1)) ELSE 0 END) +
5   (CASE c.short WHEN 'd-tlb' THEN (m.dTLB_overall - (max*0.1)) ELSE 0 END) +
6   (CASE c.short WHEN 'i-access' THEN (m.i_overall - (max*0.1)) ELSE 0 END) +
7   (CASE c.short WHEN 'i-tlb' THEN (m.iTLB_overall - (max*0.1)) ELSE 0 END) +
8   (CASE c.short WHEN 'br-i' THEN (m.branch_overall - (max*0.1)) ELSE 0 END) +
9   (CASE c.short WHEN 'fpt-fast' THEN (m.fast_FP - (max*0.1)) ELSE 0 END) +
10  (CASE c.short WHEN 'fpt-slow' THEN (m.slow_FP - (max*0.1)) ELSE 0 END))
11 AS score FROM recommendation AS r INNER JOIN rec_cat AS rc ON r.id = rc.id_rec
12 INNER JOIN category AS c ON rc.id_cat = c.id JOIN metric AS m JOIN (SELECT
13   MAX(m.L1d_hits, m.L2d_hits, m.L2d_misses, m.dTLB_overall, m.i_overall,
14   m.iTLB_overall, m.branch_overall, m.fast_FP, m.slow_FP) AS max
15   FROM metric AS m WHERE ((m.overall * 100 / (0.5 * (100 - m.ratio_fp +
16   m.ratio_fp)) > 1) AND (m.id = @RID))
17 WHERE ((r.loop <= @LPD) AND (m.hotspot_type = 'loop')) OR
18   ((r.loop IS NULL) AND (m.hotspot_type = 'function')) AND (m.id = @RID)
19 GROUP BY r.id ORDER BY score DESC;

```

Listing 1.2. Rule for selection of recommendations based on AutoSCOPE strategy

number of simultaneous streams) to select recommendations and set parameters to tune these optimizations according to the underlaying architecture.

New rules to select recommendations can be formulated, implemented as SQL queries, and stored in the database. The infrastructure will automatically take new rules into consideration next time the workflow is executed. Rules may incorporate current and future raw measurement and metrics and also be specific for a given architecture, programming language, and compiler. There is no limit neither on the number of rules to select recommendation the infrastructure can handle nor on the number of recommendations each rule returns.

3.5 Implementation of Optimizations

There is a separate Performance Optimization Implementation Tool – POIT, which is a standalone program, to implement each possible optimization. Currently POITs are built using ROSE or PIPS, but it is possible to use other tools/languages to build them. The objective of each POIT is to apply a specific recommendation from the ranked list of recommendations. To achieve this objective, it is necessary to first verify the validity of applying recommendation. Thus, each POIT implements one or more pattern checkers which may be part of the POIT or may be a separate program.

The pattern checker can be implemented using static source code analysis tools (*e.g.*, Bison/Flex, ROSE, LLVM). For example, supposing the recommendation is a modification in the source code, the POIT should verify that the pattern of the source code matches the pattern of the recommendation (*e.g.*, to apply a loop interchange the source code should have nested loops in the exact line number where the bottleneck was identified).

The POIT also has to perform a dependence analysis to insure no constraints are violated by the optimization. When the pattern checker finds a match, the POIT then proceeds to apply the recommended optimization to the application. It is possible to use other languages/tools to create POITs.

A POIT must be implemented for each additional diagnosis and recommendation that is added to the database. A reference to these new POITs must also be inserted into the database and correlated to the recommendation it implements. This way, the infrastructure will automatically take the new POITs into consideration next time the workflow is executed.

3.6 Workflow Implementation

Control of the workflow is implemented by a C program which accepts user-defined options to customize the behavior of the optimization process. To make the optimization process user-friendly, the workflow is initiated by a single command line with just few options. The control program first sets up the environment (*e.g.* temporary directory for intermediary files, debug, and log) and verifies the invoked module pre-requirements are available (*e.g.* database, compiler, application binary or source code, permissions, and external tools and libraries). If the user has chosen to manually implement the optimizations the application

binary and arguments are input. If the user has chosen the automated optimization, then the user must also provide the original source code which is compiled using the user-defined compiler and compiler flags. This phase is only required because of the modifications on the source code which may be done during the optimization phase of the process.

The infrastructure currently supports Intel, GNU, and LLVM compilers. Support for other compilers may be added. Both single source file codes and Makefile-based compilation structures are supported. The compilation phase implemented by this infrastructure also takes into consideration the use of compiler options as may be suggested by the recommendation phase of the workflow.

The control program invokes each phase of the automated optimization process including recompiling the optimized application and re-initiating the optimization cycle. The control program exits the optimization cycle by outputting any unimplemented recommendations for optimization.

4 Case Studies of Application

The case studies presented illustrate three levels of application of the PerfExpert implementation of the infrastructure using HPCToolKit as the measurement tool and the LCPI analysis for the Sandy Bridge chips of the compute nodes of the Stampede system: *a*) an implementation of the heat transfer equation where the optimization cycle was repeated implementing two optimizations, *b*) a full scale Lattice Boltzmann code where PerfExpert was able to automatically implement only one optimization, *c*) the Back Propagation [5] code from the Rodinia benchmark set and *d*) the Speckle Reducing Anisotropic Diffusion code from the Rodinia benchmark set [20]. The first three case studies use regular grids while the fourth was chosen to illustrate application to an application with an irregular grid execution behavior.

Heat Transfer: the LCPI analysis shows that L1 and L2 data misses and TLB misses dominate data access time in the main loop of the program which is an iterative solver for linear equations. The first cycle through the PerfExpert system reversed the order of the inner and outer loops while the second cycle added loop tiling for the two inner loops, improving the performance in 9% on average. The results for each of the two cycles are shown in Table 2.

Lattice Boltzmann Method: the LCPI metrics for this OpenMP implementation of the LBM shows a high cost for data accesses mostly because of L3 cache misses. MACPO analysis reported nearly zero data reuse within a loop iteration and that 23 simultaneous streams were being referenced. The recommendation for this code was to split the loop to limit the number of simultaneous streams to 6. This optimization improved the application performance in 18% on average, as shown in Table 2.

Rodinia Back Propagation: the LCPI analysis module reported that L1 and L2 misses and data TLB misses are significant contributors to data access in the `bpnn_adjust_weights` function. The recommendation for this function

was to interchange the order of two nested loops in this function. The benefit of this optimization is shown in Table 3, where the performance gain is around 30% of the execution time for serial execution. When the number of OpenMP threads increases, the performance gain decreases because of the non-parallel regions present in the code. However, the contribution to the runtime of the optimized function was reduced to a quarter of its value.

Rodinia Speckle Reducing Anisotropic Diffusion: this code has a performance issue due to poor data locality similar to the Back Propagation code but in this case, the loop which was optimized dominates the execution time. Interchanging the loop order reduced the total runtime by 38% on average.

These optimizations are ones that most compilers could make if they could resolve the code segment. In most cases (except for loop interchange in the Heat Transfer code), the code segment contained code patterns which was not resolved by the static analyses of the optimizing compilers. The codes presented in this section were compiled using the maximum optimization level (`-O3`) supported.

To illustrate how the metrics and architectural parameters are used together to select a recommendation, consider the LBM case study where the number of streams in a loop exceeds the number of simultaneous prefetcher streams the architecture can support. The optimization is to split the loop into multiple loops where the number of simultaneous streams in each loop is about the number of simultaneous streams the architecture can support. In this case, the rule `SELECT 49 AS recommendation id, 10 AS score FROM macpo_metric AS m, hound AS h WHERE (m.streams > (0.8 * h.streams)) AND (0 < @LPD)` correlates runtime metrics (from measurements) with architectural parameters. The only variable extracted from the source code is the loop depth. `h.streams` is an architectural parameters and `m.streams` is a measured characteristic of the code. 49 is the number of the recommendation to split loops.

Besides these experiments, an additional number of interesting cases were presented at an internal workshop on use of PerfExpert organized by the Flemish Supercomputer Center. In this workshop, PerfExpert was used only through the selection and recommendation phases, optimizations were implemented manually. One case, an infectious disease propagation code, was found to have a very

Table 2. Total runtime for the original and optimized versions of Heat Transfer and Lattice Boltzmann Method codes (Intel Compiler) according to the number of threads

# of Threads or Processes	Heat Transfer				Lattice Boltzmann Method		
	Original	1st Cycle	2nd Cycle	Gain	Original	Optimized	Gain
1	219.4s	222.5s	212.2s	3.6%	294.4s	241.9s	17.9%
2	110.8s	109.9s	96.3s	13.0%	177.0s	129.3s	27.0%
4	75.1s	75.3s	68.0s	9.5%	78.6s	65.7s	16.4%
8	73.6s	73.5s	64.2s	12.8%	40.6s	33.5s	17.5%
16	37.0s	37.1s	33.9s	8.4%	25.7s	22.6s	12.1%

Table 3. Total runtime and contribution of `bpnn_adjust_weights` function for the runtime on Rodinia Back Propagation according to the number of threads and compiler. Runtime value for both original and optimized codes are presented in seconds.

OpenMP Threads	Total Runtime								bpnn_adjust_weights function							
	Original		Optimized		Gain		Original		Optimized		Gain					
	gcc	icc	gcc	icc	gcc	icc	gcc	icc	gcc	icc	gcc	icc	gcc	icc	gcc	icc
1	99.2	97.9	67.1	68.7	32%	30%	40%	41%	10%	10%	76%	76%				
2	79.7	78.2	62.2	63.7	22%	19%	37%	39%	6%	6%	85%	85%				
4	66.0	66.1	56.5	57.7	14%	13%	19%	17%	3%	3%	83%	81%				
8	59.9	57.6	53.8	54.7	10%	5%	9%	10%	2%	2%	80%	81%				
16	55.5	54.7	52.4	53.6	6%	2%	6%	3%	1%	1%	77%	48%				

high fraction of data access cost from L2 and L3 misses arising from a single data structure. The code had a speedup of a factor of 3 when a data structure transformation from an array of structures was mapped to a structure of arrays.

5 Related Research

Past research related to the automation of performance analysis spans multiple domains: conceptual foundations for automation of optimization, performance optimization tools which automate one or more of the phases of performance optimization, tools needed for implementation of automated performance optimization, autotuning and profile guided compilation. This section briefly summarizes past research in each of these domains.

The only past research in conceptual foundations for automating performance optimization of which we are aware is the work of Eigenmann and co-workers, first on development of a methodology [7], on the application of the methodology to automating program parallelization [2], and then on implementation and application of the methodology [10,22,21].

There are many tools which can be used during one or more steps of the workflow presented in this paper. In regards to performance measurements, tools such as HPCToolkit [29]/PAPI [3], Intel VTune Amplifier [24], TAU [27] and Open|SpeedShop [26] provide hardware performance counter based measurements on the CPU, Intel Phi coprocessor, and NVIDIA GPUs. For parallel applications, tools like Score-P [15] and Extrae [1] provide a powerful instrumentation environment to measure the performance of MPI, OpenMP, and CUDA codes on the aforementioned processors/accelerators. To collect memory access trace, there are well-established tools like Valgrind [18] and PIN [14], however, we chose to use MACPO [23], a low overhead tool which not only gathers sample-based traces but also implements computation of metrics required for diagnosing the causes of bottlenecks and formulating optimizations.

There are several open-source tools and systems for source code verification, analysis, and manipulation. Tools like Bison and Flex[13] provide an easy method to create static checkers using grammars while tools like LLVM [12], ROSE, and PIPS provide powerful interfaces for source code analysis and manipulation.

The current implementation of the analysis and diagnosis phase presented in this paper is taken from PerfExpert and MACPO [23]. Many open-source tools for implementing source-to-source transformations (ROSE [25], PIPS [9], Rascal [11]) are available. However, integration of these tools into a workflow of runtime metric based performance optimization to implement optimizations which have been specified for localized hotspots has not been previous attempted.

There is one proprietary tool [6] which has demonstrated automation of the fourth phase (implementation of optimizations) using IBM proprietary compilers and tools using considerably different measurements and methods from those reported in this paper. The only other open-source modular infrastructure to performance measurement, analysis, and optimization of which we are aware is the Component Based Tool Framework project – CBTF [19], but it lacks the fourth step of automatic implementation of performance optimizations. There are two other research domains which automation of performance optimization overlap or are synergistic: autotuning and compilation using runtime information.

Autotuning is another method of end-to-end performance optimizations which delivers optimized source code. Autotuning is based on an entirely different approach, iterative, feedback-controlled, search of the space of possible code structures and compiler options. Therefore we do not cover autotuning research to any depth. However, recent work in autotuning such as the one found in the SUPER project [30] which uses models and patterns to guide the search, introduces similarity to the methods used in the development of performance optimization tools where the selection of optimizations is based on measurements and analyses. The infrastructure for automating performance optimization and its implementation described in this paper can in some measure be regarded as integrating the approaches of performance optimization with the approach of autotuning found on the AutoTune project [16] since it combines selection of the optimizations it implements based on runtime measurements with iterative evaluation of the performance gains due each optimization. Alternatively, the workflow described in this paper can be thought of as autotuning where each code transformation/environment setup is known to almost always lead to performance enhancement. We therefore mention some of the autotuning methods and tools which use model and/or pattern-based searches [17,31,8].

There are several compilers which have the capability to incorporate information from execution profiles and/or other measurements of program execution behavior to enhance the optimizations possible using only static source code information. This technique is sometimes abbreviated to profiler guided optimization or PGO. These include the Intel and GCC compiler suites, Microsoft Visual C++, and Oracle Solaris Studio. PGO uses the results of test runs of the instrumented program to optimize the final generated code. The compiler is used to access data from a sample run of the program across a representative

input set. The data indicates which areas of the program are executed more frequently, and which areas are executed less frequently. Optimizations benefit from profile-guided feedback because they are less reliant on heuristics when making compilation decisions. The caveat, however, is that the sample of data fed to the program during the profiling stage must be statistically representative of the typical usage scenarios; otherwise, profile-guided feedback has the potential to harm the overall performance of the final build instead of improving it. Moreover, none of the PGO capabilities gathers the comprehensive set of measurements which are gathered by good performance measurement tools. For example, on Sandy Bridge chips the LCPI analysis of PerfExpert gathers 19 performance counters and MACPO generates 7 metrics related to data structure access characteristics. The implementation of optimizations requiring source-to-source transformations described in this paper is highly similar to what compilers do when they choose to modify code to enhance performance except that the choice of optimization and its implementation incorporates full and complete information about the execution behavior of the program and is far more comprehensive in its coverage than any current compiler implemented PGO phase.

6 Future Development

We are currently extending the current set of optimizations which can be automatically implemented for the node level execution environment of the Stampede system at TACC at the University of Texas where the compute nodes are Intel Sandy Bridge chips and Intel Xeon Phi chips. These implementations are readily extended to other x86 based architectures. The intermediate term extensions will include measurement and analysis of MPI communication, OpenMP runtime system, and I/O (Lustre FS and MPI I/O), recommendation and implementation of optimization coming after. We are seeking collaborators to extend the infrastructure to automate chip/node level optimization for Atom, ARM, NVIDIA, and GPUs. We are also surveying the rapidly expanding set of tools for measurement of energy use and efficiency to determine if power as a resource is a candidate for automated optimization within our infrastructure.

7 Conclusion

We have demonstrated that automation of performance optimization/adaptation spanning multiple execution environments and resource types is a practical and realizable goal. The infrastructure presented in this paper is a concrete example of how this goal may be obtained by leveraging the power of the many existing tools for the several independent phases of performance optimization.

Earlier versions of PerfExpert have been downloaded and installed on many systems in several countries. PerfExpert 4, which is implemented in the infrastructure presented in this paper, is the now standard tool to analyze and optimize performance on TACC systems. Recently, several HPC systems in Belgium have installed PerfExpert on their systems and plan to use it as the standard tool to optimize applications performance.

PerfExpert 4 for Intel Sandy Bridge and Xeon Phi chips is publicly available for download at <http://www.tacc.utexas.edu/perfexpert>. The package includes the entire modular infrastructure which can be customized to other execution environments.

References

1. Alonso, P., Badia, R.M., Labarta, J., Barreda, M., Dolz, M.F., Mayo, R., Quintana-Orti, E.S., Reyes, R.: Tools for Power-Energy Modelling and Analysis of Parallel Scientific Applications. In: Proceedings of the International Conference on Parallel Processing, pp. 420–429 (2012)
2. Banerjee, U., Eigenmann, R., Nicolau, A., Padua, D.A.: Automatic Program Parallelization. Proceedings of the IEEE 81(2), 211–243 (1993)
3. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A Portable Programming Interface for Performance Evaluation on Modern Processors. International Journal of High Performance Computing Applications 14(3), 189–204 (2000)
4. Burtscher, M., Kim, B.-D., Diamond, J., McCalpin, J.D., Koesterke, L., Browne, J.: PerfExpert: An Easy-to-Use Performance Diagnosis Tool for HPC Applications. In: Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (2010)
5. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.-H., Skadron, K.: Rodinia: A Benchmark Suite for Heterogeneous Computing. In: Proceedings of the IEEE International Symposium on Workload Characterization, pp. 44–54 (2009)
6. Chung, I.H., Cong, G., Klepacki, D., Sbaraglia, S., Seelam, S., Wen, H.F.: A Framework for Automated Performance Bottleneck Detection. In: Proceedings of the IEEE International Symposium on Parallel and Distributed processing (2008)
7. Eigenmann, R.: Toward a Methodology of Optimizing Programs for High-Performance Computers. In: Proceedings of the International Conference on Supercomputing, pp. 27–36 (1993)
8. Huck, K.A., Malony, A.D., Shende, S., Morris, A.: Knowledge Support and Automation for Performance Analysis with PerfExplorer 2.0. Large-Scale Programming Tools and Environments. Special Issue of Scientific Programming 16(2-3), 123–134 (2008)
9. Keryell, R., Ancourt, C., Coelho, F., Creusillet, B., Irigoin, F.: PIPS: a Workbench for Building Interprocedural Parallelizers, Compilers and Optimizers. Technical report, École Nationale Supérieure des Mines de Paris (1996)
10. Kim, S.W., Park, I., Eigenmann, R.: A Performance Advisor Tool for Shared-Memory Parallel Programming. In: Midkiff, S.P., Moreira, J.E., Gupta, M., Chatterjee, S., Ferrante, J., Prins, J.F., Pugh, B., Tseng, C.-W. (eds.) LCPC 2000. LNCS, vol. 2017, pp. 274–288. Springer, Heidelberg (2001)
11. Klint, P., van der Storm, T., Vinju, J.: RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In: Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation, pp. 168–177 (2009)
12. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proceedings of the International Symposium on Code Generation and Optimization, pp. 75–86 (2004)
13. Llc, B.: Parser Generators. Books LLC. Wiki Series (2010)
14. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, vol. 40(6), pp. 190–200 (2005)

15. Mey, D.A., Biersdorf, S., Bischof, C., Diethelm, K., Eschweiler, D., Gerndt, M., Knüpfer, A., Lorenz, D., Malony, A., Nagel, W.E., Oleynik, Y., Rössel, C., Saviankou, P., Schmidl, D., Shende, S., Wagner, M., Wesarg, B., Wolf, F.: Score-P: A Unified Performance Measurement System for Petascale Applications. In: Proceedings of the International Conference on Competence in High Performance Computing, pp. 85–97 (2011)
16. Miceli, R., et al.: AutoTune: A Plugin-Driven Approach to the Automatic Tuning of Parallel Applications. In: Manninen, P., Öster, P. (eds.) PARA 2012. LNCS, vol. 7782, pp. 328–342. Springer, Heidelberg (2013)
17. Miller, B.P., Callaghan, M.D., Cargille, J.M., Hollingsworth, J.K., Irvin, R.B., Karavanic, K.L., Kunchithapadam, K., Newhall, T.: The Paradyn Parallel Performance Measurement Tool. IEEE Computer 28(11), 37–46 (1995)
18. Nethercote, N., Seward, J.: Valgrind: A Program Supervision Framework. Electronic Notes in Theoretical Computer Science 89(2), 44–66 (2003)
19. Online, <http://ft.ornl.gov/doku/cbtfw/>
20. Online, <https://www.cs.virginia.edu/~skadron/wiki/rodinia/>
21. Pan, Z., Armstrong, B., Bae, H., Eigenmann, R.: On the Interaction of Tiling and Automatic Parallelization. In: Mueller, M.S., Chapman, B.M., de Supinski, B.R., Malony, A.D., Voss, M. (eds.) IWOMP 2005/IWOMP 2006. LNCS, vol. 4315, pp. 24–35. Springer, Heidelberg (2008)
22. Park, I., Kapadia, N.H., Figueiredo, R.J., Eigenmann, R., Fortes, J.A.B.: Towards an Integrated, Web-executable Parallel Programming Tool Environment. In: Proceedings of the Supercomputing Conference (2000)
23. Rane, A., Browne, J.: Enhancing Performance Optimization of Multicore Chips and Multichip Nodes with Data Structure Metrics. In: Proceedings of the Int. Conference on Parallel Architectures and Compilation Techniques, pp. 147–156 (2012)
24. Reinders, J.: VTune Performance Analyzer Essentials, 1st edn. Intel Press (2005)
25. Schordan, M., Quinlan, D.: A Source-To-Source Architecture for User-Defined Optimizations. In: Böszörnyi, L., Schojer, P. (eds.) JMLC 2003. LNCS, vol. 2789, pp. 214–223. Springer, Heidelberg (2003)
26. Schulz, M., Galarowicz, J., Maghrak, D., Hachfeld, W.: Open|SpeedShop: An open source infrastructure for parallel performance analysis. Scientific Programming 16(2-3), 105–121 (2008)
27. Shende, S., Malony, A.D.: The Tau Parallel Performance System. International Journal of High Performance Computing Applications 20(2), 287–311 (2006)
28. Sopeju, O.A., Burtscher, M., Rane, A., Browne, J.: AutoSCOPE: Automatic Suggestions for Code Optimizations using PerfExpert. In: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, pp. 19–25 (2011)
29. Tallent, N., Mellor-Crummey, J., Adhianto, L., Fagan, M., Krentel, M.: HPC-Toolkit: performance tools for scientific computing. Journal of Physics: Conference Series 125(1) (2008)
30. Tiwari, A., Chen, C., Chame, J., Hall, M., Hollingsworth, J.K.: A Scalable Auto-tuning Framework for Compiler Optimization. In: Proceedings of the IEEE Symposium on Parallel and Distributed Processing (2009)
31. Wen, H., Sbaraglia, S., Seelam, S., Chung, I., Cong, G., Klepacki, D.: A Productivity Centered Tools Framework for Application Performance Tuning. In: Proceedings of the International Conference on the Quantitative Evaluation of Systems, pp. 273–274 (2007)

Designing MPI Library with Dynamic Connected Transport (DCT) of InfiniBand: Early Experiences*

Hari Subramoni, Khaled Hamidouche, Akshey Venkatesh,
Sourav Chakraborty, and Dhabaleswar K. Panda

Department of Computer Science and Engineering, The Ohio State University,
Columbus, OH, USA

{subramoni.1, hamidouche.2, akshayvenkatesh.1,
chakraborty.52}@osu.edu, {panda}@cse.ohio-state.edu

Abstract. The Dynamic Connected (DC) InfiniBand transport protocol has recently been introduced by Mellanox to address several shortcomings of the older Reliable Connection (RC), eXtended Reliable Connection (XRC), and Unreliable Datagram (UD) transport protocols. DC aims to support all of the features provided by RC — such as RDMA, atomics, and hardware reliability — while allowing processes to communicate with any remote process with just one DC queue pair (QP), like UD. In this paper we present the salient features of the new DC protocol including its connection and communication models. We design new verbs-level collective benchmarks to study the behavior of the new DC transport and understand the performance / memory trade-offs it presents. We then use this knowledge to propose multiple designs for MPI over DC. We evaluate an implementation of our design in the MVAPICH2 MPI library using standard MPI benchmarks and applications. To the best of our knowledge, this is the first such design of an MPI library over the new DC transport. Our experimental results at the microbenchmark level show that the DC-based design in MVAPICH2 is able to deliver 42% and 43% improvement in latency for large message All-to-one exchanges over XRC and RC respectively. DC-based designs are also able to give 20% and 8% improvement for small message One-to-all exchanges over RC and XRC respectively. For the All-to-all communication pattern, DC is able to deliver performance comparable to RC/XRC while outperforming in memory consumption. At the application level, for NAMD on 620 processes, the DC-based designs in MVAPICH2 outperform designs based on RC, XRC, and UD by 22%, 10%, and 13% respectively in execution time. With DL-POLY, DC outperforms RC and XRC by 75% and 30%, respectively, in total completion time while delivering performance similar to UD.

Keywords: Dynamic Connected Transport, InfiniBand, High Performance Computing, Network technology.

1 Introduction

Two key drivers fueling the growth of supercomputers over the last decade are the current trends in multi-/many-core architectures and the availability of commodity,

* This research is supported in part by National Science Foundation grants #OCI-0926691, #OCI-1148371, #CCF-1213084, and #CNS-1347189.

RDMA-enabled, and high-performance interconnects such as InfiniBand [1]. As supercomputing systems head to exascale levels, the core densities of current generation multi-/many-core architectures are expected to increase manyfold. Consequently, the amount of memory available per core on such systems is expected to be low compared to current generation systems. Thus, the focus is being increasingly given to reducing the memory footprint of applications and communication middlewares that targets these systems.

MPI [2] is a popular programming model for parallel scientific applications running on current generation multi-petaflop supercomputers. As such, the MPI library design is crucial in supporting high-performance and scalable communication for applications on these large-scale clusters. Over the last decade, InfiniBand has become an increasingly popular interconnect for deploying modern supercomputing systems. InfiniBand offers several communication protocols like Reliable Connection (RC), eXtended Reliable Connection (XRC), and Unreliable Datagram (UD) which all have different performance and memory characteristics. Several implementations of MPI over InfiniBand like MVAPICH2 [3], Open MPI [4], and Intel MPI [5] are already delivering good performance for applications on these systems using the RC transport. While RC delivers the best performance at a small scale, earlier work has shown that the RC transport requires several Kilobytes of memory per connected peer, leading to significant memory usage and performance degradation at large-scale runs [6].

To alleviate this, MPI libraries like MVAPICH2 take advantage of protocols like XRC and UD to reduce the memory footprint while delivering comparable or better performance [7, 6]. UD-based solutions enable the MPI library to keep the memory required for creating communication queue pairs (QPs) constant. However, implementing MPI over UD requires software-based segmentation, ordering, and re-transmission within the MPI library. Furthermore, UD transport does not support several novel features like hardware-based atomic operations. XRC-based solutions, on the other hand, avoid these software overheads and also have hardware support for atomic operations. However, the QP memory footprint scales linearly with the number compute nodes. With exascale systems expected to have O(100,000) to O(1,000,000) nodes [8], this presents a huge overhead in terms of memory.

Recently, Mellanox has introduced the Dynamic Connected (DC) transport protocol. The DC transport attempts to give the same feature set of RC while providing UD-like flexibility in terms of communicating with all peers using one QP. Given this capability, the connection memory required for DC can potentially reduce by a factor equal to the *number cores per node * number of nodes* compared to RC. Similarly, the connection memory required can reduce by a factor of *number of nodes* compared to XRC. These represent potentially large savings as system size and core densities continue to increase. In this paper, we present the salient features of the new DC protocol. We provide a comparison of the connection model of DC with other IB transport protocols. We compare and contrast the connection model of DC with other IB transport protocols. We design a micro-benchmark suite at the verbs [9] level to study the performance, and memory tradeoffs of the new DC transport. We then use this knowledge to propose a dynamically adaptive scheme which uses multiple designs for implementing MPI over DC. We implement our designs in the MVAPICH2 MPI library and evaluate

it using standard MPI benchmarks as well as applications. Additionally, memory usage is also measured. To the best of our knowledge this is the first such study of the DC protocol and design of MPI over DC. To summarize, this paper makes the following contributions:

- Understanding the behavior of Mellanox DC transport protocol for different messaging patterns and designing an MPI library to make efficient use this protocol with multiple design variations
- Providing an analysis of the benefits and shortcomings of DC-based MPI operations and comparing with other state of the art protocols
- Showing the memory, performance, and scalability benefits with micro-benchmarks and applications

Experimental results at the microbenchmark level show that the DC-based design in MVAPICH2 is able to deliver 42% and 43% improvement in latency for large message All-to-one exchanges over XRC, and RC respectively. DC-based designs are also able to give 20% and 8% improvement for small message One-to-all exchanges over RC and XRC respectively. For the All-to-all communication pattern, DC is able to deliver performance comparable to RC/XRC while outperforming them in memory consumption. At the application level, for NAMD on 620 processes, the DC-based designs in MVAPICH2 outperform designs based on RC, XRC, and UD by 22%, 10%, and 13%, respectively, in execution time. With DL-POLY, DC outperforms RC and XRC by 75% and 30% respectively in total completion time while delivering performance similar to UD.

2 InfiniBand Transport Services

InfiniBand is a popular switched interconnect fabric used by 41% of the Top500 Supercomputing systems [10]. InfiniBand Architecture [9] defines a switched network fabric for interconnecting processing and I/O nodes, using a queue-based model. It supports two communication semantics: Channel Semantics (Send-Receive communication) over RC, XRC, and UD; and Memory Semantics (Remote Direct Memory Access communication) over RC and XRC. Both semantics can perform zero-copy transfers from source-to-destination buffers without additional host-level memory copies. RC is connection-oriented and requires dedicated QP for destination processes while the connection-less UD transport uses a single QP for all [6, 11]. XRC optimizes QP allocation by requiring each process to create only one QP per node [7].

3 Dynamic Connected (DC) Transport

In the following sections, we describe the connection model, the communication model, and the destination addressing scheme used for DC.

3.1 Connection Model

Figure 1 depicts the connection models for the different transport protocols InfiniBand supports for a fully-connected job. Each node has two processes that are fully connected to the all processes on other nodes. We do not account for intra-node IB connections since the focus of this paper is on MPI and MPI libraries generally use a shared memory channel for communication within a node instead of network loopback. For generality, we assume that the cluster has N nodes with C cores per node in all equations.

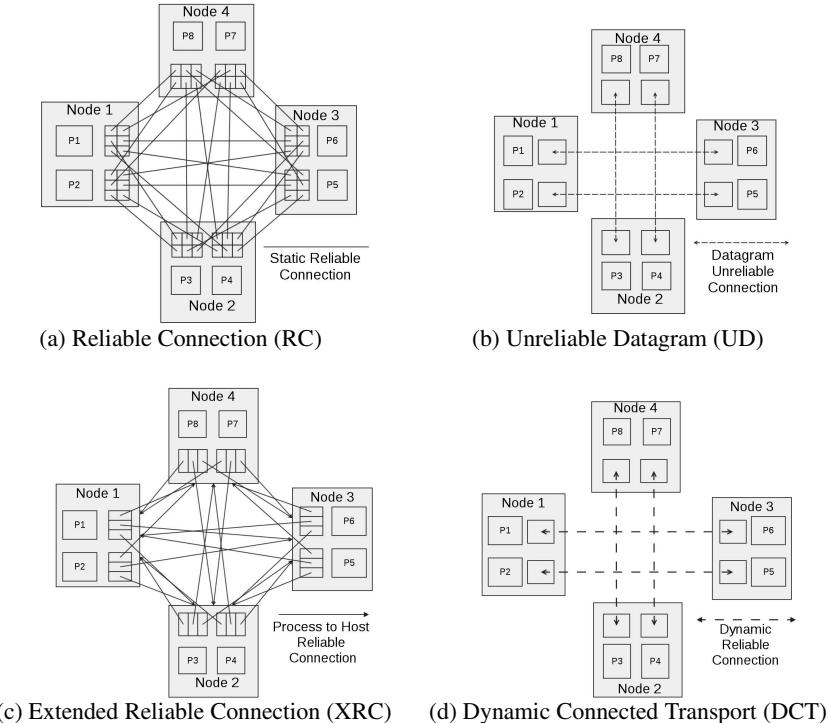


Fig. 1. Connection models for different transport protocols in InfiniBand

Figure 1a depicts the connection model for RC. To maintain full connectivity in a cluster, each process must have $(N - 1)*C$ QPs. Figure 1c shows a fully-connected XRC job. Instead of requiring a new QP for each process, now each process needs to only have one QP per node to be fully connected. In the best case the number of QPs required for a fully-connected job in a cluster, is only N QPs. This reduces the number of QPs required by a factor of C . However, on next-generation exascale systems where the number of hosts is expected to reach O(100,000) to O(1,000,000) nodes [8], this still presents a huge overhead in terms of memory. Figures 1b and 1d represent the connection model for UD and DC respectively. As we can see, the number of QPs

required for a fully-connected job reduces to a constant value irrespective of the number of processes in the job. This presents a very compelling reason to use these protocols to design next-generation communication middlewares.

3.2 Communication Objects, Addressing Scheme and Communication Model

Communication Objects: The main communication objects used in DC are DC-Initiators (DCINI) and DC Targets (DCTGT). DCINIs are analogous to the send QPs used in other IB protocols. Any process that has to transmit data to a peer using DC must create a DCINI for this purpose. Processes can use DCINIs as they would use a UD QP, i.e. 1) it can be used to transmit data to any peer as seen in Section 3.1 and 2) we can transition the state of the DCINI without providing the information about the remote QP. DCTGTs are analogous to the receive QPs. But unlike RC and UD where the use of a Shared Receive Queue (SRQ) [12] is optional, the DCTGTs *must* be backed by an SRQ.

Addressing Scheme: Each DCTGT will be assigned a number called the *dct_number* by the IB HCA. This value will be unique to the HCA. Thus one can uniquely identify a DCTGT on the network by a combination of the Local Identifier (LID) that is unique to the IB subnet and the DCTGT number. As we saw before, a DCINI does not require the target information while transitioning to the Ready To Send (RTS) state. Hence, the protocol uses a combination of an address handler field and the number of the remote DCTGT to route the packet to the correct destination. The address handler field contains the remote LID to aid in the routing decision in the network. An additional parameter known as the *dc_key* must be provided to all the DC objects to enable communication. This parameter must be same across all processes that wish to communicate with each other.

Communication Model: Figure 2 depicts the communication model used by DC to establish / tear-down connections and progress communication. As described above, the first step is for the processes to create DCINIs / DCTGT and transition them to the appropriate state. Once this is done, the processes are ready to send and receive data. Let's say Process-2 (P2) enqueues WQEs to send two data items to Process-1 (P1) and one data item to Process-3 (P3). The HCA at P2 will first send out a connect message in an *inline* fashion to P1 ahead of sending the actual data. On receiving the *Connect* message, the target HCA temporarily allocates a context to receive the incoming data. Once the data arrives, it looks up the DCTGT number and places it in the SRQ specified by the user when the DCTGT was created. The connection is kept alive by the target HCA in anticipation of receiving further messages from the same sender. If such messages are not received, the target times out and releases the context thereby breaking the connection. In this case, due to the back-to-back nature of the sends from P2, the connection will still be alive when the second packet arrives. Thus the HCA at P1 places the data in the appropriate SRQ as before. As P2 is using the same DCINI to send data to both P1 and P3, it will issue a *Disconnect* message to P1 as soon as it receives the ACK for the last packet from P1. We use the term *sender-initiated disconnect* to refer to this behavior in the rest of the paper. P2 will then send a *Connect* message in an

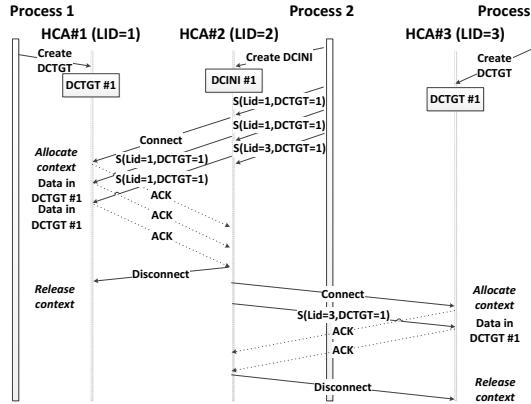


Fig. 2. Communication model(Courtesy [13])

inline fashion as before to P3 followed by the actual data. The HCA at P3 follows the same pattern as the HCA at P1 for the first message it received, as depicted in Figure 2.

4 Understanding Basic DC Performance

We use several verbs-level benchmarks in order to understand the performance tradeoffs associated with using DC protocol. We first use verbs-level point-to-point latency and bandwidth benchmarks to understand and analyze the performance one can obtain by using the DC protocol. We do not show numbers for the XRC protocol as the behavior will be similar to the RC protocol for small systems sizes. Notably, the verbs-level benchmarks only measure latency and bandwidth using the UD protocol up to 2,048 bytes. Beyond this the time to perform segmentation and re-assembly of packets must be taken into account to accurately measure latency and bandwidth using UD.

Figure 3a compares the verbs-level, point-to-point latency observed with RC, DC, and UD protocols for different message sizes. As seen, RC gives the best latency for small messages. We can also observe that DC adds an additional 100 nanoseconds of latency on top of RC. We believe that this is due to overhead involved with establishing connections for each send operation. Due to the ping-pong nature of the benchmark, the DCTGT will get timed out after each send operation, thereby breaking the connection and forcing the DCINI to establish connection with each send operation. With large messages, this overhead is not significant as the time taken for transmitting the data is much greater than the overhead of connection establishment. Figure 3b depicts the verbs-level, point-to-point bandwidth obtained for RC, DC, and UD protocols. As we can see, the RC delivers peak performance for all message ranges. DC is able to deliver performance on par with RC for small and large messages. However, it performs worse than RC in the medium message range. We are still investigating the reasons behind this performance drop.

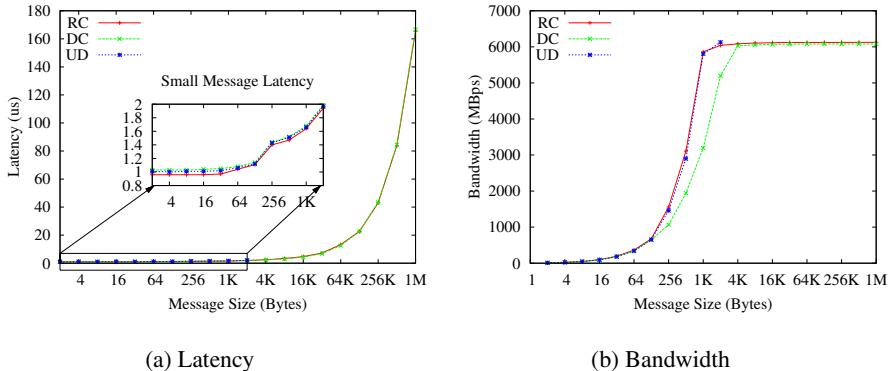


Fig. 3. Verbs-level point-to-point performance

Next we evaluate two collective communication patterns that are common in scientific applications - *One-to-all* and *All-to-one*. We design and develop two new verbs-level benchmarks based on the basic point-to-point, verbs-level latency benchmark for this purpose. Due to the overheads of maintaining reliability and flow control, we do not enable the use of UD protocol for this benchmark. As we saw in Section 3.2, the use of just one DCINI has the potential to create significant serialization at the sender side. To evaluate the impact of this serialization on the performance of One-to-all and All-to-one communication patterns, we run experiments with DC in two modes - 1) sender using only one DCINI to perform all send operations (indicated by DC-One-QP) and 2) sender using separate DCINI per communicating peer for communication (indicated by DC-One-QP-Per-Peer). Figure 4 compares the performance of RC, DC-One-QP-Per-Peer, and DC-One-QP for the One-to-all pattern. As we can see, with an increasing number of peers, the performance of DC-One-QP for small messages degrades significantly. On the other hand, DC-One-QP-Per-Peer is able to deliver performance comparable to RC. Next we evaluate the performance of the All-to-one communication pattern with RC, DC-One-QP and DC-One-QP-Per-Peer. Figure 5 depicts the performance of the All-to-one benchmark with increasing number of peers. As we can see, both DC-One-QP and DC-One-QP-Per-Peer perform slightly better than RC for small messages. For all other message sizes, we do not observe any significant difference in performance between the RC and the two DC variants.

In summary, we observe four major trends with DC - 1) DC adds an overhead of 100 nanoseconds when compared with RC if there are frequent disconnections, 2) One-to-all performance of small messages can be affected significantly if the sender is only using one DCINI, 3) DC performs better for All-to-one pattern with small messages, and 4) the number of DCINIs does not affect the performance of All-to-one communication pattern. Another side effect of serialization when using just one DCINI can come in the form of head-of-line blocking. This can happen if a small message destined for peer A gets blocked by an ongoing large message transmission to peer B on the same DCINI.

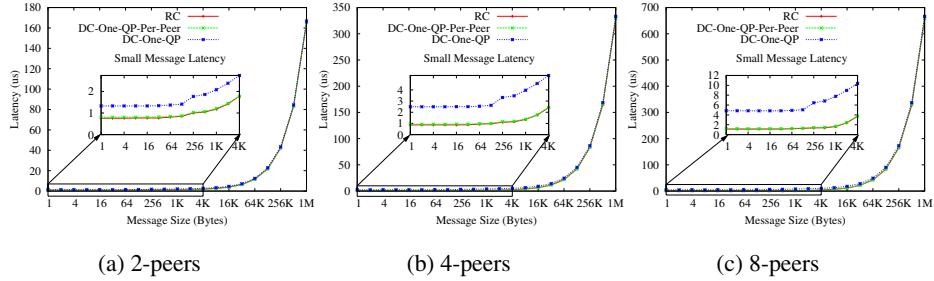


Fig. 4. Verbs-level One-to-all latency

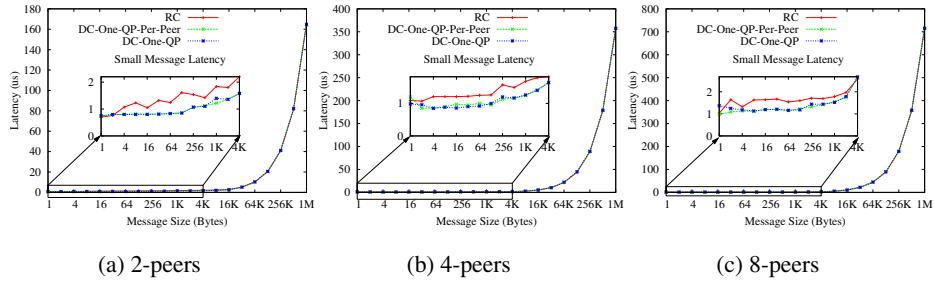


Fig. 5. Verbs-level All-to-one latency

5 Design of DC-Aware MPI Library

As we saw in Section 4, there are several design challenges that need to be overcome to create a high performance MPI library that uses DC. The design needs to achieve four major goals:

1. Avoid the negative effects of serialization with one DCINI
 2. Avoid the performance penalty of sender-initiated connection tear down
 3. Reduce the communication memory footprint of the MPI library and,
 4. Avoid head-of-line blocking with DCINIs where small messages may get blocked behind an ongoing large message transfer to a different peer

Notably the overhead of connection tear down due to the sender having no data to send is a separate issue and needs to be handled at the HCA. With these goals in mind, we propose three different DC-aware designs for the MVAPICH2 MPI library.

Design-1 (DC-E-RC): As we saw in Figures 4 and 5, by using one DCINI per peer, DC-One-QP-Per-Peer is able to perform on par with RC. Hence, the simplest approach would be to create a design where we dedicate one DCINI per communicating peer. We call this design “DC-Emulating-RC” or *DC-E-RC* in short. As we are dedicating one DCINI per peer, we will avoid the negative effects of serialization. We will also avoid

the performance penalty of sender-initiated disconnects as one DCINI will only be used to send data to one peer. Thus, using *DC-E-RC*, we will be able to achieve goals 1, 2, and 4 that we identified above.

Design-2 (*DC-E-UD*): Although the One-to-all communication pattern requires multiple DCINIs to be created to avoid the effects of serialization, the basic point-to-point communication pattern, as well as the All-to-one pattern, performed equally well with DC-One-QP. Hence, depending on the communication pattern and the number of processes involved, it is possible that we may not require multiple DCINIs. However, using just one DCINI for small and large messages can cause head-of-line blocking we identified in Section 4. Hence we use two DCINIs - one for small messages and one for large messages to avoid this. Furthermore, using just two DCINIs will present us with significant savings in memory especially as the scale increases. Since we emulate the UD transport protocol by using just one set of QPs, we term this “DC-Emulating-UD” or *DC-E-UD* in short. Thus, using *DC-E-UD*, we achieve goals 3 and 4. We may also achieve goals 1 and 2 depending on the communication pattern.

Design-3 (*DC-Pool*): As noted above, the *DC-E-RC* design has the potential to alleviate the negative effects of serialization at the DCINI. However, it will cause a huge bloat in memory due to the over-provisioning of DCINIs. *DC-E-UD* on the other hand will reduce the memory footprint but may fail to alleviate the effects of serialization for all communication patterns. Hence, an approach that takes a middle path between these two extreme scenarios may be able to achieve the first three goals. With this in mind, we propose the third hybrid scheme. Here, we use a pool of DCINIs for sending data. When a process wants to send data, it picks a free DCINI from the pool, transmits data and then returns it back to the pool. As we choose a different DCINI for each transmission, it can avoid the negative effects of serialization we identified in Section 4. Furthermore, by limiting the number of DCINIs created to a small value, we also limit the communication memory footprint of the MPI library to a constant value. Thus, we will be able to achieve goals 1 and 3 with this design.

However, due to the fact that the DCINIs get used for both large and small message transmission, head-of-line blocking scenarios may still occur. Furthermore, as each send uses a separate DCINI, it is virtually guaranteed that sender-initiated disconnects will happen with each send operation even if subsequent sends are to the same peer. To remedy these two issues, we propose two optimizations to the pool-based approach: 1) We split our DCINI pool into two - one for small messages and one for large messages. Thus we ensure that a small message never gets blocked by a large message transfer to a different peer. 2) Once a process selects a DCINI for performing communication with a peer, it stores the information of the DCINI in the Virtual Channel (VC) data structure called VC table (VCT) maintained in MVAPICH2. The VC table is used to keep track of the current state of communication with each peer a process has. For future transmissions to the same peer, the process will just re-use the same DCINI. By doing this, we avoid sender-initiated disconnects from happening if a process is continually communicating with a particular peer. We call this hybrid design as *DC*.

Figure 6 depicts how the *DC-Pool* design is implemented inside the MVAPICH2 MPI library. MVAPICH2 uses an *on-demand* method to establish connections. In this

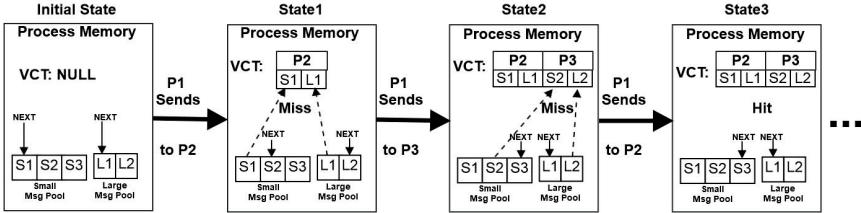


Fig. 6. DC-Pool-based design

method, a connection is established with a peer only if a process needs to communicate with it. Hence at the initial state (just after job startup), the VC table would be empty or NULL. We create two pools of DCINIs - one for small messages and one for large messages. We also initialize *NEXT* pointers that point to next DCINI that needs to be used in the pool. The DCINIs are selected in a round robin fashion to distribute the load. From this initial state, lets assume that the sender process (P1) wants to communicate with P2 (represented by State 1 in Figure 6). It will check the VC table to see if a communication channel already exists for that process. If it does not, it is considered as a “Miss”. It will then retrieve the next small message and large message DCINIs from the pool and store the information of the DCINIs in the entry of VC table for the peer. It will also increment the *NEXT* pointers for both pools. The next time P1 wants to communicate with P2, it will retrieve the DCINI information directly from the VC table entry for P2 instead of using a new DCINI from the pool (represented by State 3). State 2 in Figure 6 depicts the state of the various data structures after P1 establishes connection with a new peer P3.

6 Experimental Results

In this section, we describe the experimental setup used to conduct micro-benchmark and application experiments to evaluate the efficacy of existing transport protocols and that of the proposed DC designs. An in-depth analysis of the results are also provided to correlate design motivations and observed behavior. All results reported here are averages of multiple runs to discard the effect of system noise.

6.1 Experimental Setup

The setup consists of 32 Ivy Bridge Compute nodes interconnected by Mellanox FDR switch SX6036. The Intel Ivy Bridge processors consist of Xeon dual ten-core sockets operating at 2.80 GHz with 32 GB RAM. Each node is equipped with MT4113 Connect-IB HCAs (54 Gbps data rate) with PCI-Ex Gen3 interfaces. The operating system used is RHEL 6.2 with kernel version 2.6.32-220.el6, and Mellanox OpenFabrics version 2.1-1.0.0. All micro-benchmark results have been obtained through the use of OSU Micro-Benchmark suite (OMB) [14].

6.2 MPI Level Point-to-Point Results

In this section, we show the influence of transport protocol choice on the performance of communication between either a pair or multiple pairs of MPI processes through micro-benchmarks.

In Figure 7a, the effect of protocol choice on the latency of communicating messages is shown as a function of message size. As this experiment involves a single communicating pair, each process from a pair uses a maximum of one QP and hence the performance of DC-Pool, DC-E-UD, and DC-E-RC are on par. Further, any QP thrashing effects can be ruled out. It can be observed that the performance of RC in the short message range outperforms that of DC-Pool by around 100 nanoseconds. We attribute this degradation to connection re-establishment after the implicit teardown that occurs upon transmission of a message and the subsequent timeout that occurs when using DC-Pool. RC, on the other hand, maintains the connection until explicit teardown instruction and hence shows the best latency. Also, it is to be noted that use of UD transport mechanism has an overhead in comparison with the rest of the available mechanisms for the entire message range. This is an expected result owing to the combination of UD being a connection-less transport, being unreliable and in the large message range, suffering from packetization owing to MTU limits (on current Mellanox hardware, MTU is 2KB).

In Figure 7b, the effect of protocol choice on the bandwidth of communicating messages between a single pair of MPI process is shown as a function of message size. The experiment involves posting multiple non-blocking send operations and waiting for their completion. It can be seen in the small message range that the bandwidth of DC-Pool, DC-E-UD, and DC-E-RC are on par with RC. This behavior is explained by noting that connection teardown occurs in DC if messages are not posted to the DC QP within a certain timeout interval. Within the short message range, this is an unlikely event for the bandwidth experiment due to multiple sends being posted and the relatively short amount of time needed to transfer each of them. However, for the rest of the range DC-Pool, DC-E-UD, and DC-E-RC show marginally smaller bandwidth in comparison with RC and XRC. This can be attributed to the effect of connection teardown that are likely to occur during the progression of medium and large message transfers with DC protocol owing to the larger latency involved in their transmission.

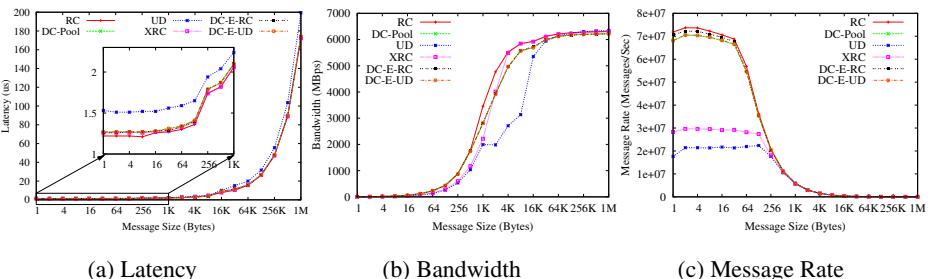


Fig. 7. MPI-level point-to-point performance

Lastly, in Figure 7c, the effect of protocol choice on the messaging rate between 20 pairs of MPI processes (20 processes on one node) is shown as a function of message size. As expected, due to connection teardown overheads, DC-Pool and its variants have marginally smaller messaging rates in comparison with RC. In the small message range, DC-E-RC performs slightly better than DC-E-UD or DC-Pool. From the point of view of the node HCA, the aggregate number of QPs to be managed increases from 20 to 40 as transport protocol is switched DC-E-RC from DC-E-UD or DC-Pool. For small messages, subsequent cache thrashing leads to performance degradation in case of DC-E-UD or DC-Pool.

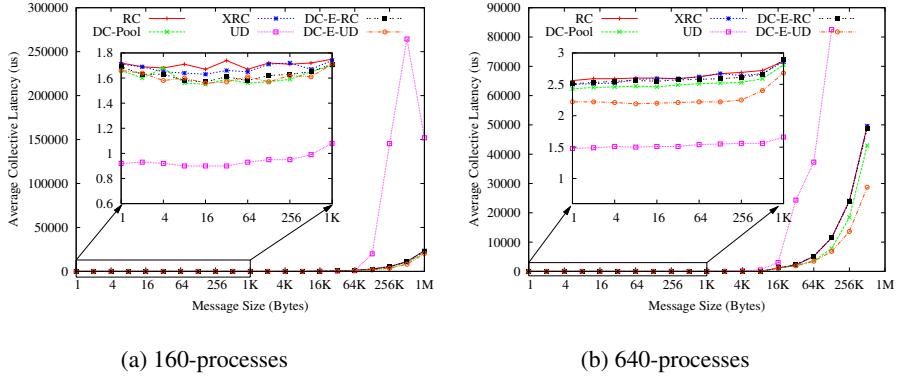
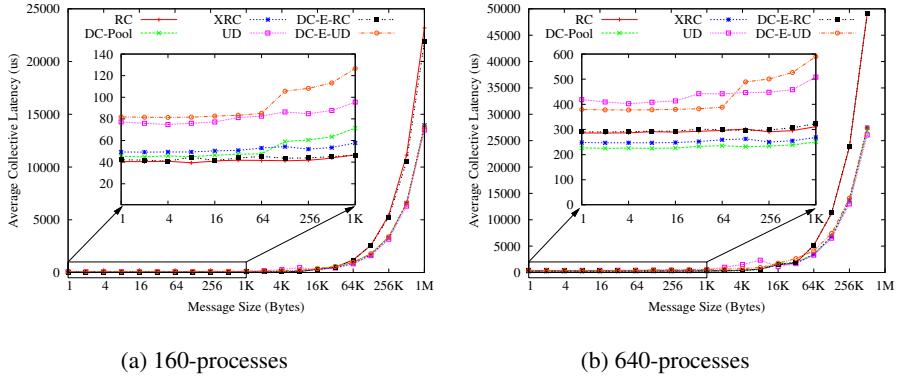
6.3 MPI Level Collective Results

MPI collectives predominantly rely on MPI point-to-point operations as their base primitives. As seen in Section 6.5, transport selection has an effect on performance on MPI point-to-point primitives and in this section we depict the effect of transport protocol choice on the performance of MPI collective communication patterns such as All-to-one, One-to-all, and All-to-all. All the collective experiments have been conducted with either 160, 320, or 640 processes to study the effect on scalability as well. All the figures depict the average of latencies experienced by all processes. Due to lack of space and the repetitive nature of the figures, we drop the results for 320 processes and just show results for 160 and 640 processes.

In Figures 8a and 8b, the performance of All-to-one collective is shown. For all message ranges, the root process receives a message directly from every other process involved in the collective. In the short message range, it can be seen that UD followed by DC-E-UD achieve the shortest latency among all the protocols. To understand this, the mechanism undertaken by the root must be considered. With RC/DC-Pool/XRC schemes, when the HCA at the root receives a message from a non-root, it has the overhead of sending back an ACK before processing the next message which is avoided in a UD-based scheme. However, beyond a message size of 2 KB, explicit software packetization and reliability overheads add to the latency and cause the latency to deteriorate. DC-E-UD performs relatively well in the entire message range owing to two reasons. First, it avoids cache thrashing experienced by RC, DC-E-RC, and XRC. Second, the serialization effects are avoided at a receiving DC QP, further bringing down the average latency of the All-to-one operation.

In Figures 9a and 9b, the performance of an One-to-all collective is shown. For all message ranges, the root process sends a message directly to every other process involved in the collective. The trends here are inverse of that noted for the All-to-one collective. In the short message range, the use of UD and DC-E-UD causes serialization due to the use of a single QP and hence these two schemes have the greatest latencies among all the protocols. Beyond the 16KB message range, RC and DC-E-RC schemes suffer from QP cache thrashing at the HCA which causes a significant increase in the average latency in comparison to the UD, XRC, and DC-E-UD.

Finally, Figures 10a and 10b depict the performance of the All-to-all personalized collective. The operation involves pairs of MPI processes taking turns to communicate with each other. Hence, it is clear that the performance of the operation largely hinges on the performance of the basic point-to-point primitive performance of underlying

**Fig. 8.** MPI-level All-to-one performance**Fig. 9.** MPI-level One-to-all performance

transport protocols. As seen Section 6.5, UD is the only transport mechanism that shows considerable deviation from the rest of the protocols and the effect of this shows up in Figures 10a and 10b. At around the 16KB mark, UD starts showing a considerable increase in point-to-point latency (Figure 7a) and so do the All-to-all trends. It is to be noted that the abrupt peak and dip in case of UD is a result of switch from degraded packetized sends to zero copy UD rendezvous transfers [15].

6.4 Memory Footprint

Figure 11 contrasts the memory consumed by the MPI library on QP resources for different protocols. It can be seen that RC and DC-E-RC schemes consume the most memory as each process sets up dedicated QPs for every other process in the MPI job. UD scheme uses a single QP but has a constant overhead for rendezvous-protocol based

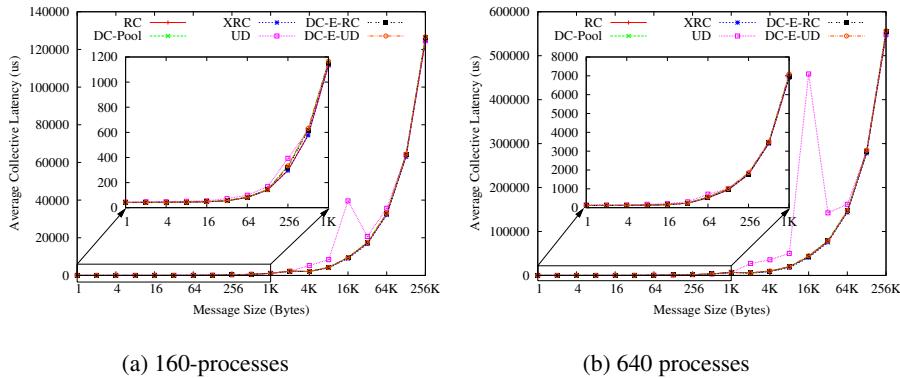


Fig. 10. MPI-level All-to-all performance

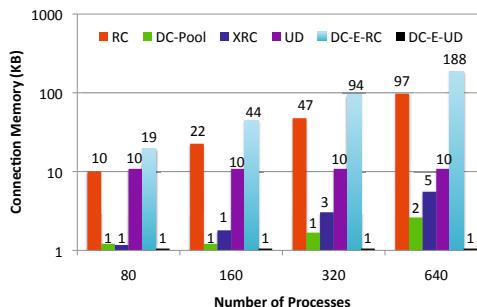


Fig. 11. Memory Footprint of Transport Protocols

transmissions. Under the XRC scheme, each process has a QP for every node and hence has a factor of core-count-per-node reduction in memory consumption in comparison to RC. The DC-Pool scheme has a further reduced memory footprint due to the use of minimal set of QPs. As DC-E-UD emulates UD scheme, there are only two DCINIs used for all connections and hence memory consumption remains constant.

6.5 Application Level Results

In this section we evaluate the different transport modes using two applications. These are more likely to model real-world use than microbenchmarks. We evaluate using the two molecular dynamics application NAMD and the DL-POLY.

NAMD: is a fully-featured, production molecular dynamics program for high performance simulation of large bimolecular systems [16]. NAMD is based on Charm++ parallel objects, which is a machine independent parallel programming system. The parallel decomposition strategy used by NAMD is a three-dimensional patchwork which

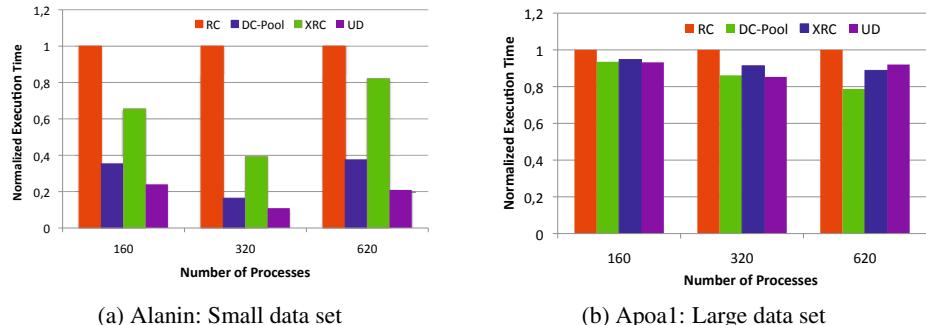


Fig. 12. NAMD application: Normalized execution time

involves a near-neighbors communication pattern. Of the standard data sets available for use with NAMD, we use the *apoA1* and *alanin* datasets.

DL-POLY: is a general purpose classical molecular dynamics (MD) simulation software developed at Daresbury Laboratory [17]. It spends 27% and 17% of its communication time which is almost 40% of the execution time on MPI_Allreduce and MPI_Send/Recv respectively. We used two standard data sets: the *TEST1: Sodium chloride* (27,000 ions) and the *TEST2: Sodium chloride* (216,000 ions).

Figures 12a and 12b show the normalized execution time of NAMD on three different systems sizes (160, 320 and, 620 processes) using the *alanin* (small) and *apoA1* (large) data sets respectively. With *alanin* DC-Pool outperforms both RC and XRC modes by 84% and 25% respectively with 320 processes configuration, and by 63%, 35% with 620 processes configuration. With the three system size configurations, UD is performing the best as *alanin* is a small data set and hence the communications patterns use small message sizes. With large data set size, as shown in Figure 12b, DC-Pool is performing the best with 620 processes configuration. Indeed DC-Pool outperforms RC, XRC and UD by 22%, 10%, and 13% respectively.

Normalized execution time of DL-POLY with TEST1 and TEST2 is depicted in Figures 13a and 13b respectively. DL-POLY is enable to run at large scale system (600 cores), with a small data set (TEST1) as each process need to have a minimum of data to initialize. As shown in Figure 13, DC-Pool is performing better than RC and XRC by up to 77% and 13% respectively on 320 processes configuration. As with NAMD, for DL-POLY with small data set, UD is performing better than DC-Pool by 9% on the 320 processes configuration. With large data set (TEST2), DC-Pool is performing the best for 160 processes configuration. On 600 processes, DC-Pool shows 75% and 30% improvement compared to RC and XRC and almost the same performance than UD (4% benefits for UD). We are trying to understand the communication characteristics of DL-POLY better to analyze the reason for the benefit seen with UD.

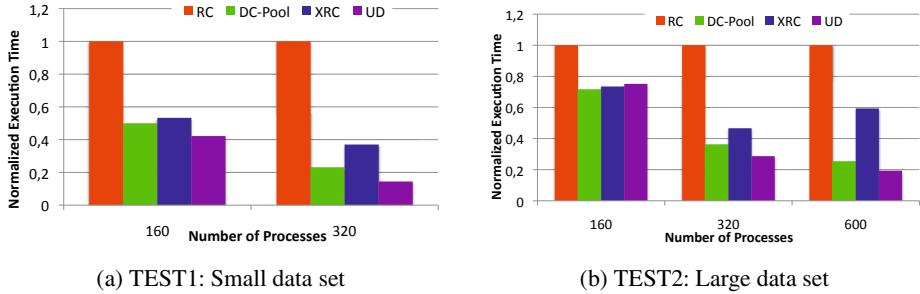


Fig. 13. DL-POLY application: Normalized execution time

7 Related Work

Scalability of MPI libraries over InfiniBand has been the focus of many recent research works. Koop, et al. proposed [6] and later improved [15] a connection-less, design based on the UD protocol. Similar designs have been proposed for iWARP as well. [18]. Hardware-based multicast over UD has been shown to improve performance of collectives in [19–21]. A hybrid scheme where UD was used for first 16 messages before setting up a RC connection was presented by Yu, et al [22]. The effects of different transport services on WAN was analyzed in [23]. A dynamic approach that uses both RC and UD protocols based on the communication pattern of the application has also been designed [11]. Using SRQ to reduce requirement of communication context has been studied in [24]. Shipman, et al. proposed using different SRQs for different data sizes to improve utilization of communication buffers on RC [25]. Erimli [26] used a method based on multiply-linked lists to share memory for WQEs. Connect-X, the first HCA to support XRC was evaluated by Sur, et al [24]. Koop, et al. explored design of MPI library, performance and memory footprint of XRC protocol in [7].

8 Conclusion and Future Work

In this paper we presented the salient features of the new DC protocol including its connection and communication models. We designed new verbs-level collective benchmarks to study the behavior of the new DC transport and understand the performance / memory trade-offs it presents. We then used this knowledge to propose multiple designs for MPI over DC. An implementation of our design in the MVAPICH2 MPI library was evaluated using standard MPI benchmarks and applications. To the best of our knowledge, this is the first such design of an MPI library over the new DC transport. Our experimental results at the microbenchmark level showed that the DC based design in MVAPICH2 was able to deliver 42% and 43% improvement in latency for large message All-to-one exchanges over XRC and RC respectively. DC based designs were also able to give 20% and 8% improvement for small message One-to-all exchanges over RC and XRC respectively. For the All-to-all communication pattern, DC was able

to deliver performance comparable to RC/XRC while outperforming in memory consumption. At the application level, for NAMD on 620 processes, the DC based designs in MVAPICH2 outperformed designs based on RC, XRC and UD by 22%, 10%, and 13% respectively. With DL-POLY, DC outperformed RC and XRC by 75%, and 30% respectively while delivering performance similar to UD.

References

1. InfiniBand Trade Association, <http://www.infinibandta.com>
2. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard (March 1994)
3. Panda, D.K., Tomko, K., Schulz, K., Majumdar, A.: The MVAPICH Project: Evolution and Sustainability of an Open Source Production Quality MPI Library for HPC. In: Int'l Workshop on Sustainable Software for Science: Practice and Experiences, Held in Conjunction with Int'l Conference on Supercomputing, SC 2013 (November 2013)
4. The Open MPI Development Team: Open MPI: Open Source High Performance Computing, <http://www.open-mpi.org>
5. Intel Corporation: Intel MPI Library, <http://software.intel.com/en-us/intel-mpi-library/>
6. Koop, M.J., Sur, S., Gao, Q., Panda, D.K.: High Performance MPI Design using Unreliable Datagram for Ultra-scale InfiniBand Clusters. In: ICS 2007: Proceedings of the 21st Annual International Conference on Supercomputing, pp. 180–189. ACM, New York (2007)
7. Koop, M., Sridhar, J., Panda, D.K.: Scalable MPI Design over InfiniBand using eXtended Reliable Connection. IEEE Int'l Conference on Cluster Computing (Cluster 2008) (September 2008)
8. Kogge, P.: ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems
9. InfiniBand Trade Association: InfiniBand Architecture Specification 1, Release 1.0, <http://www.infinibandta.com>
10. Meuer, H., Strohmaier, E., Dongarra, J., Simon, H.: TOP 500 Supercomputer Sites, <http://www.top500.org>
11. Koop, M.J., Jones, T., Panda, D.K.: MVAPICH-Aptus: Scalable high-performance multi-transport MPI over InfiniBand. In: IPDPS 2008, pp. 1–12 (2008)
12. Sur, S., Chai, L., Jin, H.-W., Panda, D.K.: Shared Receive Queue Based Scalable MPI Design for InfiniBand Clusters. In: Proceedings of the 20th International Conference on Parallel and Distributed Processing, IPDPS 2006, p. 101. IEEE Computer Society, Washington, DC (2006)
13. Crupnicoff, D., Kagan, M., Shahar, A., Bloch, N., Chapman, H.: Dynamically Connected Transport Service (July 3, 2012), US Patent 8,213,315
14. Network Based Computing Laboratory: OSU Micro-benchmarks, <http://mvapich.cse.ohio-state.edu/benchmarks/>
15. Koop, M.J., Sur, S., Panda, D.K.: Zero-copy Protocol for MPI using InfiniBand Unreliable Datagram. In: CLUSTER 2007: Proceedings of the 2007 IEEE International Conference on Cluster Computing, pp. 179–186. IEEE Computer Society, Washington, DC (2007)
16. Phillips, J.C., Braun, R., Wang, W., Gumbart, J., Tajkhorshid, E., Villa, E., Chipot, C., Skeel, R.D., Kale, L., Schulten, K.: Scalable Molecular Dynamics with NAMD. Journal of computational chemistry 26(16), 1781–1802 (2005)
17. Forester, T., Smith, W.: DL-POLY Package of Molecular Simulation. CCLRC, Daresbury Laboratory: Daresbury, Warrington, England (1996)

18. Rashti, M.J., Grant, R.E., Afsahi, A., Balaji, P.: iWARP redefined: Scalable connectionless communication over high-speed Ethernet. In: 2010 International Conference on High Performance Computing (HiPC), pp. 1–10. IEEE (2010)
19. Liu, J., Wu, J., Kini, S.P., Wyckoff, P., Panda, D.K.: High Performance RDMA-Based MPI Implementation over InfiniBand. In: 17th Annual ACM International Conference on Supercomputing (June 2003)
20. Mamidala, A., Liu, J., Panda, D.K.: Efficient Barrier and Allreduce on IBA clusters using hardware multicast and adaptive algorithms. In: IEEE Cluster Computing (2004)
21. Mamidala, A.R., Narravula, S., Vishnu, A., Santhanaraman, G., Panda, D.K.: On Using Connection-Oriented Vs. Connection-Less Transport for Performance and Scalability of Collective and One-Sided Operations: Trade-offs and Impact. In: Proceedings of the 12th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, pp. 46–54. ACM (2007)
22. Yu, W., Gao, Q., Panda, D.K.: Adaptive Connection Management for Scalable MPI over InfiniBand. In: International Parallel and Distributed Processing Symposium, IPDPS (2006)
23. Yu, W., Rao, N.S., Vetter, J.S.: Experimental Analysis of InfiniBand Transport Services on WAN. In: International Conference on Networking, Architecture, and Storage, NAS 2008, pp. 233–240 (2008)
24. Sur, S., Chai, L., Jin, H.W., Panda, D.K.: Shared Receive Queue based Scalable MPI Design for InfiniBand Clusters. In: International Parallel and Distributed Processing Symposium, IPDPS (2006)
25. Shipman, G.M., Woodall, T.S., Graham, R.L., Maccabe, A.B., Bridges, P.G.: InfiniBand Scalability in Open MPI. In: 20th International Parallel and Distributed Processing Symposium, IPDPS 2006, p. 10. IEEE (2006)
26. Erimli, B.: Arrangement in an InfiniBand Channel Adapter for Sharing Memory Space for Work Queue Entries using Multiply-linked Lists (March18, 2008) US Patent 7,346,707

RADAR: Runtime Asymmetric Data-Access Driven Scientific Data Replication

John Jenkins^{1,2,*}, Xiaocheng Zou¹, Houjun Tang¹, Dries Kimpe²,
Robert Ross², and Nagiza F. Samatova^{1,3}

¹ North Carolina State University, Raleigh, NC 27695, USA

{xzou2, htang4}@ncsu.edu,

samatova@csc.ncsu.edu, jenkins@mcs.anl.gov

² Argonne National Laboratory, Argonne, IL 60439, USA

{jenkins, kimpe, rrross}@mcs.anl.gov

³ Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA

Abstract. Efficient I/O on large-scale spatiotemporal scientific data requires scrutiny of both the logical layout of the data (e.g., row-major vs. column-major) and the physical layout (e.g., distribution on parallel filesystems). For increasingly complex datasets, hand optimization is a difficult matter prone to error and not scalable to the increasing heterogeneity of analysis workloads. Given these factors, we present a partial data replication system called RADAR. We capture datatype- and collective-aware I/O access patterns (indicating logical access) via MPI-IO tracing and use a combination of coarse-grained and fine-grained performance modeling to evaluate and select optimized physical data distributions for the task at hand. Unlike conventional methods, we store all replica data and metadata, along with the original untouched data, under a single file container using the object abstraction in parallel filesystems. Our system results in many-fold improvements in some commonly used subvolume decomposition access patterns. Moreover, the modeling approach can determine whether such optimizations should be undertaken in the first place.

1 Introduction

In high-performance computing (HPC) systems and parallel filesystems such as PVFS [7], Lustre [38], and GPFS [37], choosing a “good” distribution of data across multiple storage devices is a difficult problem, which numerous works have been dedicated to solving. The combination of high dimensionality (multiple variables distributed in a spatiotemporal domain) and distributed requests over many processes complicates making an informed decision about how to place data in order to achieve high performance. The problem is exacerbated when noncontiguous access patterns are induced on storage, such as subvolume access. Optimizations made to reduce or eliminate noncontiguous disk access, such as two-phase collective I/O [45], create new access patterns for which the data distribution may not be optimized, compounding the problem of selecting a good distribution.

* Corresponding author.

Previous works have looked at data layout optimization in an HPC context in two general respects: modifying the logical layout of data (i.e., the mapping of data into a linear address space), with the goal of producing specialized data organizations (e.g., data layout using space-filling curves [26,35], range-query processing on scientific data [14,13,21,25], multiresolution analysis [10,22]) and optimizing the physical distribution of the logical address space onto storage devices to better match the mapping of process requests to I/O servers [42,43,57]. However, these works have some combination of the following potential problems: modified logical formats introduce both interoperability concerns and difficulties related to manual management of the custom format; works that provide multiple data layouts or replicate data in multiple formats rely on creating potentially hidden directories/files for each, leading to a large number of files to process any time the dataset is used; and the distribution formats are either fixed or made to optimize for a specific access characteristic (e.g., disk thrashing via DiskSim [3], requests to a single segment of file).

To mitigate these problems, we present a model-driven, adaptive layout optimization framework, called RADAR (Runtime Asymmetric Data-Access driven scientific data Replication), using object-based filesystem semantics. Our layout optimization is based on partial replication, allowing a controllable increase in dataset sizes in exchange for I/O performance optimization. Furthermore, as opposed to previous works, which fix either the regions of data to replicate or the replication format, we allow variability in both. In other words, we choose both the data to replicate and the layout of the replicated data in distributed storage. We specifically focus on read-time I/O optimization, with the goal of optimizing for commonly used access patterns on a per-dataset basis. We present the following contributions:

1) Adaptive replica management policy. Given a set of I/O access patterns, our replica layout manager (Section 2.1) uses an I/O performance modeling approach to (1) create replicas with varied layouts for performance optimization of input access patterns and (2) to rank replicas for inclusion under storage-limited scenarios. Furthermore, our approach can gracefully handle imbalances in both server loads and client loads, using performance modeling to account for the former and distribution heuristics to account for the latter. Using a prototype MPI-IO driver, we show that our method is effective at accelerating common subvolume decomposition tasks, showing multifold speedups under many scenarios.

2) Single-container, nonintrusive dataset storage. All replica data and metadata are stored alongside the original, unchanged data in a single file container, achieved through direct management of objects on an object-storage system (Section 2.2). Distribution of replica data among a fixed set of replica objects is enabled through a combination of sparse-file capabilities and a object-slice-based allocation scheme.

3) Datatype- and collective-aware MPI-IO tracing. As an enabling technology, we develop a tracer capable of collecting full logical I/O requests with low overhead at the MPI-IO-level (Section 2.3). It is configurable to collect either precollective or postcollective optimizations (or both).

Our paper is organized as follows. Section 2 describes the framework, including necessary background. Section 3 presents our experiments. Section 4 examines related work. Section 5 briefly summarizes our conclusions.

2 Method

The system workflow is realized by client-side, middleware, and filesystem components, as shown in Figure 1. We gather access information through a datatype-aware, collective-aware I/O trace layer. These individual process traces are processed by using a pattern analyzer, outputting access patterns of interest, such as strided access. Our replica layout manager ingests these patterns and determines what data to replicate and in what format. Our replica-aware, object-storage-based ADIO implementation matches I/O requests to replications, redirecting the subsequent object operations to the EOF direct-object interface of PVFS (see Section 2.2).

Since the bulk of our methodology lies in the layout manager and works independently of the method of replica distribution and access pattern generation, we first discuss it in Section 2.1. Next, we describe the data management policies employed by our method in Section 2.2, followed by the tracing and trace-analysis components in Section 2.3. The replica-aware ADIO driver is discussed in Section 2.4.

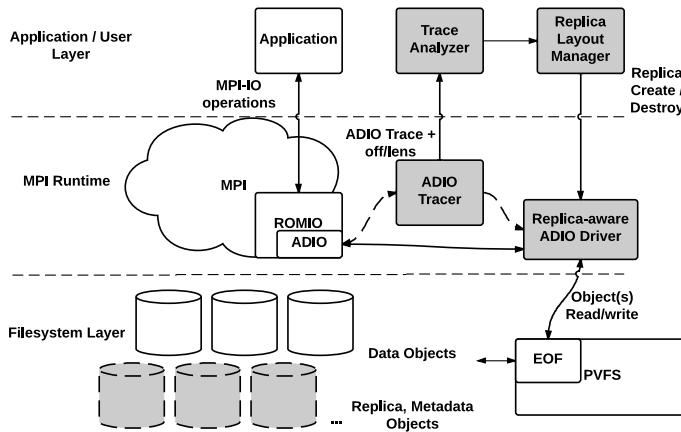


Fig. 1. RADAR components, across the I/O software stack. The shaded figures delineate our contributions.

2.1 RADAR Layout Manager

The goal of RADAR’s layout manager component is to create replicas with a layout that improves I/O performance given a set of access patterns. To provide a framework capable of doing so, we use two strategies. First, to optimize in the presence of concurrent accesses, we generate replicas for *time-delimited pattern sets*. Second, to quickly generate and evaluate candidate replica sets, we adopt a *two-phase* performance-modeling approach, using a coarse-grained performance model to quickly produce and select candidate replica sets and using a fine-grained performance model to compute the estimated performance difference against the original data layout. The following sections discuss the individual components. Temporal considerations, such as garbage-collecting unused replicas, will be a focus of future work. Furthermore, note that the layout manager

works on a per-application level and does not have a full system-level view of activity. Such an approach would require full storage system control and is out of the scope of our work.

Pattern Preprocessing. Preprocessing of the patterns is performed to remove temporal considerations from the optimization process. This strategy has been examined in previous work by converting all accesses into a log structure [1,57]; here we are looking at flexible distribution among multiple servers for read, rather than write, optimization.

We partition the patterns into *buckets*, each corresponding to a time window of length δ , using timestamps from the tracer. Each bucket is then considered a single entity for the purposes of performance modeling and optimization. Since each bucket need not be sorted in our method, the overall process is linear in the number of patterns.

Replica Layout Generation and Ranking – Performance Modeling. We use a simple, constant-time performance model to generate candidate replica layouts and a more involved model to give a relative performance comparison between the original data layout and candidate layouts. This design decision is made in order to quickly generate regularly-structured layouts for testing while retaining the ability to recognize imbalances in client and server loads.

Preliminaries. Our models use latency/bandwidth measurements over both network and storage, assuming serialization of requests at the node level (both client and server). Table 1 shows the relevant variables. We make a few simplifying assumptions across both models that, while harmful to general-purpose high-accuracy performance prediction, still allow us to make valid measures for comparative purposes over time-gated accesses in a manner that is computationally reasonable. In particular, we assume no pipelining of network and storage operations, giving a pessimistic view of system capabilities, and represent resource contention through the aggregation of request latencies based on round robin processing of request chunks.

Coarse-Grained Model. The coarse-grained model is a generalization and extension of the cost model created by Song et al. [42] to optimize accesses under uniform, synchronized accesses and uniform physical distribution corresponding to PVFS data layouts. This model, while not created for general-purpose I/O modeling, has proven useful for HPC applications with regular access patterns and is appropriate for driving our replica placement method, given that we are in full control of replica placement and can produce such regular accesses. Unlike the basis model, we additionally incorporate scheduling-based resource contention.

The cost model by Song et al. has four separate costs that are summed to find the final result: T_e , the total network latency, T_x , the total network transfer time, T_s , the total storage latency, and T_{rw} , the total read/write time. Concurrency in the model is captured by considering the number of serialized client-to-server requests and server-to-client responses. The original model computes these costs based on a number of specific, fixed layouts mapping process requests to servers (see [42] for more details). We observe that

Table 1. Performance Model Variables

System parameters	
n	I/O servers
ℓ_{net}	I/O request (network) latency
b_{net}	Network per-byte transfer time
ℓ_{sto}	I/O request (disk) latency
b_{sto}	Storage per-byte transfer time
r_s	Local storage readahead
S_b	Request buffer size used by I/O scheduler
Per-pattern-set inputs	
\mathbb{P}	Set of access patterns with process mappings
p	I/O participants (clients)
m	I/O participants per node
Coarse-grained model parameters	
n_p	Servers contacted per client (input)
B	Total request size across all patterns (derived)
r_B	Avg. request size per client per server = $B/(pn_p)$ (derived)
r_m	Avg. number of requests per client node = mn_p (derived)
r_n	Avg. number of requests per server node = $\lceil pn_p/n \rceil$ (derived)

the parameter being varied across each of the models is the *servers contacted per client*. Making this an explicit variable n_p allows us to collapse the equations into the set shown in Equation 1.

$$\begin{aligned}
T_e &= (2 \times \max(r_m, r_n) + \text{sched})\ell_{net} \\
T_x &= \max(r_m, r_n)r_B b_{net} \\
T_s &= (r_n + \text{sched})\ell_{sto} \\
T_{rw} &= r_n r_B b_{sto} \\
\text{sched} &= r_n(\lceil \frac{S_b}{r_B} \rceil - 1) \quad \text{if } r_n > 1 \\
&= 0 \quad \text{otherwise}
\end{aligned} \tag{1}$$

The calculations of T_x and T_{rw} are simple, capturing the concurrent rounds of network transfer and storage transfer, respectively. T_e and T_s compute latency based on two quantities: the number of serialized requests and the number of scheduling buffers to fill, assuming a round-robin scheduling policy.

Coarse-Grained Model Usage. We derive a simple replica creation process, using the following strategy: assume the underlying accesses are regular and uniform, compute $\min_{n_p}(T_e + T_x + T_s + T_{rw})$, and then resolve any load balances by over/underprovisioning replica striping across the servers – i.e. control the access concurrency through the number of servers used for placement. After computing B and an average m , simply calculate model values for $n_p \in \{1, 2, \dots, n\}$, and choose the minimum. Next, perform the logical striping under the assumption that r is the actual request size per client. Then, compute for each pattern which servers its data resides on. An example mapping is shown in Figure 2. This layout heuristic causes more servers to

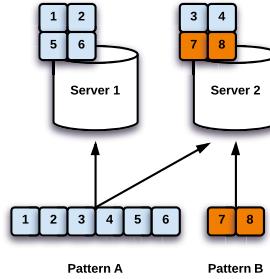


Fig. 2. Access pattern over and under provisioning based on model optimization on balanced accesses (for $n_p = 2$)

be used in the striping of accesses with larger sizes, while causing less servers to be used for smaller sizes. This is used in an attempt to balance the degree of I/O concurrency, and ultimately improve the performance, for unbalanced access patterns with large variations in size per client. Additionally, this heuristic optimizes striping as expected for patterns with uniform access sizes.

Fine-Grained Model. The fine-grained model is based on the coarse-grained model, using latency/bandwidth modeling at both the network and the storage levels to generate an overall cost and introducing resource contention through a basic scheduling mechanism. Unlike the coarse-grained model however, we need a more robust model capable of capturing load imbalance. Our approach consists of the following two steps, with the underlying steps of mapping each pattern in the pattern set to its respective client process/node and set of contacted servers: (1) compute a localized T'_e , T'_s and T'_{rw} for each server, and (2) calculate the total time to receipt T'_t for each client node based on the server calculations, with the maximum among them being the total request time.

The computation of T'_s and T'_{rw} is relatively straightforward, with T'_s being the number of noncontiguous blocks (measured at a page granularity and taking into account readahead) accessed times the storage access latency and T'_{rw} being the total size of all requests to the server times the inverse of the bandwidth. For T'_{rw} , we adjust the performance to account for readahead: if two consecutive requests are within a readahead window (default 128 KB on Linux), then the disk latency cost is avoided at the cost of consuming the bytes separating the two requests. The computation of T'_e also resembles that of the coarse-grain model, except that it also takes into account load imbalances induced by the access patterns.

The computation of T'_t is subtly different from the summation of the four components in the coarse-grain model in that, for each client node, it takes the maximum completion time of the network and storage operations across all servers that the node accesses. In particular, the transmission time back to the client uses the server's full load to simulate the client node's data being scheduled after (or in round-robin with) the remaining data the server is processing, representing a worst case schedule from the client node's point-of-view.

2.2 EOF Data Management

Background – PVFS and EOF. Recently, the “end-of-files” (EOF) [15] extension to the Parallel Virtual File System (PVFS) [7] was created to expose the object storage abstraction directly into the client space, presenting a file as a set of distinct, physically distributed objects that applications forward requests directly to, as opposed to a more implicit mapping via a striping function. Each object has an independent address space, potentially simplifying the layout of complex datasets. For example, dataset metadata can be forwarded to a single object, while the data itself can be assigned distinct objects based on timesteps, variables, and so forth. We use EOF to achieve our layout optimization goals.

RADAR Object Layout. The EOF object layout is such that nearly all RADAR data components exist under a single filename, summarized in Figure 3. The original dataset is stored in unchanged form, striped by some distribution across multiple objects. We include a file metadata object since distribution in EOF is relegated to the user. For RADAR, at file create time we allocate a number of replica objects equal to the number of data objects. We do this both for practicality reasons (limits on per-file concurrency) and for semantic reasons (currently, EOF cannot dynamically add or remove objects from a file container). A replica metadata object is used to store the mapping of replicas to object locations. We place the set of generated access patterns processed by RADAR in a dedicated object for the results to persist across multiple application runs. Note that the trace and trace analysis output currently exist outside the MPI/EOF file container, although these can be integrated with ease.

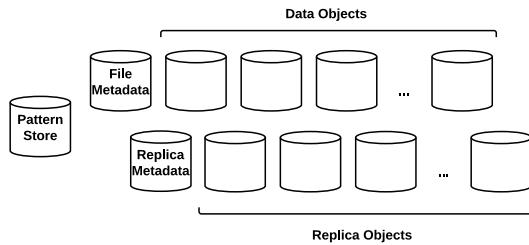


Fig. 3. EOF object layout for RADAR

Replica Object Storage Strategy. One problem with using a shared set of replica objects is how to distribute multiple replicas with heterogeneous distributions in the manner the layout manager instructs. To this end, we exploit sparse-file capabilities in the local filesystems employed by PVFS. Essentially, sparse files do not store blocks not written to aside from metadata—a block can represent data at, for example, a gigabyte offset without additionally having blocks representing all previous offsets.

Given our sparse-file strategy, we divide all objects into allocation units we call *object domains* (ODs), an example of which is shown in Figure 4. An OD corresponds to the full set of replica objects, spanning a per-object address space with fixed, large-granularity sizes. Each replicated region is placed within a single OD. In order to avoid biasing replica placement toward one object or another, replicas are assigned starting objects in round-robin order. In the figure, for example, the replica shown begins striping at the leftmost object, while the next replica created will begin striping at the next leftmost object.

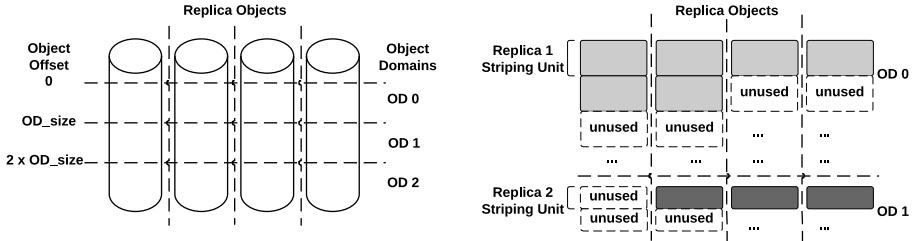


Fig. 4. *Left:* mapping of RADAR replication objects to allocation units (“object domains”). *Right:* replica placement within object domains.

2.3 I/O Tracer and Analyzer

Our tracing methodology is achieved through MPI-IO, implemented by using the underlying ADIO [46] interface in ROMIO [47], the MPI-IO implementation in MPICH. This strategy allows efficient access to both MPI datatypes used in I/O calls and the result of underlying collective calls. The output of the tracer includes all ADIO calls, as well as all per-process offset/length pairs, in a plain-text format. Compression methodologies such as inline pattern analysis or off-the-shelf compressors will be explored in future work. Note that while our method optimizes on a per-I/O-client basis, the ability to gather access information after collective optimizations have been made allows RADAR to be of potential use there as well, although additionally optimizing communication patterns is out of the scope of this work.

To gather the desired access patterns, we built a variant of the IOSig trace analysis software [4,55]. We similarly use a template matching approach; but since our tracer works at the ADIO level and additionally processes datatypes, the processing of the traces has been rewritten. For more discussion pertaining to access pattern categorization and discovery, see [4,55]. For this paper, the specific access patterns we gather are contiguous access patterns (sequential accesses with fixed or average request sizes) and k -d strided access patterns (accesses that differ in offset by a fixed value, or stride) over the logical/file view of the data, both of which are common in HPC I/O workloads, which typically corresponds to accesses along spatiotemporal domains or across multiple variables.

2.4 Replica-Aware ADIO Driver

The responsibilities of the RADAR ADIO driver are to interface with EOF, maintain the various sets of data/replica/metadata objects, and remap I/O requests into the replica space, as appropriate. Aside from the remapping portion, the rest of the processing is relatively simple, corresponding to loading/distributing file and replica metadata and driving some RADAR-specific operations, such as carrying out the replication.

Given a set of replicated data layouts and a set of I/O requests (e.g., logical file offset/length pairs via a call to `MPI_File_read`), reading from replicas occurs in two steps: finding applicable replicas to read and choosing among the resulting candidates.

When finding replicas that can satisfy a given request, a linear-time matching approach over all replicas is undesirable. Furthermore, using spatial data structures such as interval or R-trees requires flattening strided patterns into their individual contiguous blocks, leading to extremely large search structure sizes. Hence, we use a binning approach, similar to generating an *inverted index* [53,58] over the file, mapping logical regions of file (the bins) to a list of replicas overlapping with the regions to prune the search space. Additionally, to help prevent worst-case linear behavior (all replicas overlap with a bin), we employ a one-element history, under the assumption that consecutive I/O requests are highly likely to map to the same distinct replicas multiple times. Note that for read requests, we consider only replicas that contain the full file offset/length requested, since splitting up the requests further to disparate physical locations would likely reduce performance.

Once we have a set of candidate replicas to choose from, the next task is to select which replica among the set to read from. Note that this choice is nonexistent for writes, since all replicas would need to be updated. Furthermore, complicating the decision is that the information available is inherently local—individual processes cannot have a full system view. Therefore, we use a simple heuristic we call *smallest containing block* (SCB). The idea behind SCB is that of *specialization*: we consider replicas with a finer granularity in terms of contiguous chunk size to be more specialized than those with a coarser granularity, and we believe they should be prioritized in the replica selection process. Generally, then, replicas over strided data will more than likely be selected over replicas over contiguous data, since each of the strided data structures will be more sparse.

3 Experimental Evaluation

All experiments were run on the Fusion cluster at Argonne National Laboratory. Each node contains two quad-core Intel Xeon processors at 2.53 GHz with 32 GB RAM, and nodes are connected by InfiniBand QDR. Each node in Fusion contains local hard-disk storage (250 GB IBM iDataPlex). Additionally, our implementation of RADAR is based on MPICH 3.0.2 and PVFS2 2.8.1, patched with EOF. Because of issues with InfiniBand support for PVFS on Fusion, both MPI communication and PVFS client-server communication are performed via TCP over InfiniBand.

Since we use a modified version of PVFS and since each node in Fusion has local storage, we assign a subset of the nodes to serve as PVFS I/O servers and use the remaining as I/O clients. We use eight I/O servers in all experiments. A larger scale

would be desirable, although we have tried to match our performance benchmark sizes to the available resources in order to have a reasonable representation of relative I/O performance.

Table 2 shows the performance model parameters we gathered via microbenchmarks on Fusion. We use the BMI `pingpong` utility in PVFS to gather network performance through PVFS, where BMI (Buffered Message Interface) is PVFS’s client/server communication interface [8]. We use simple read benchmarking via collocating a PVFS client with a server to gather storage performance parameters. Note that the microbenchmark result for disk bandwidth is much lower than expected: the bandwidth when not going through PVFS is approximately 90 MB/s (measured through `dd`). We were unable to eliminate this discrepancy, but we believe it to be a result of internal threading and buffering overheads on the PVFS server.

Table 2. Performance Model Variables

System parameters		
n	8	I/O servers
r_s	128 KB	Local storage readahead
ℓ_{net}	$32.9\mu s$	I/O request (network) latency
ℓ_{sto}	$6.20ms$	I/O request (disk) latency
b_{net}	$0.00112\mu s$ (867 MB/s)	Network per-byte transfer time
b_{sto}	$0.0212\mu s$ (44.98 MB/s)	Storage per-byte transfer time

For our inverted list acceleration structure for replica lookup, we divided the file into 1,024 bins, each covering a 64 MB extent of data.

3.1 Benchmarks

We evaluate our layout optimization work within the context of four different multidimensional array decompositions, shown in Figure 5: row-wise (distribute volume by contiguous plane), column-wise (distribute volume by non-contiguous plane), cube-wise (distribute volume by 3D subvolume), and timestep-wise (distribute a single subvolume from a range of timesteps). The row-wise decomposition induces contiguous patterns at each process, while the remaining decompositions induce multidimensional strided patterns at each process. For all experiments, we used a subvolume of (time, X, Y, Z) dimensions $128 \times 256 \times 256 \times 256$ in row-major order, each element of which is a 32-byte structure (e.g., four C `doubles`). The total size of this dataset is 64 GB. Note that these access patterns are a superset of the access patterns exhibited by several well-known benchmarks such as MPI-Tile-I/O [34], IOR [20], and PIO-bench [39], all of which perform accesses with regular (single- or multidimensional) strides.

3.2 Decomposition Performance

The goal here is to explore the types of RADAR-generated layouts that can result in improved performance; that is, we perform the replication regardless of the fine-grained model comparison, although we still use the coarse-grained model to generate

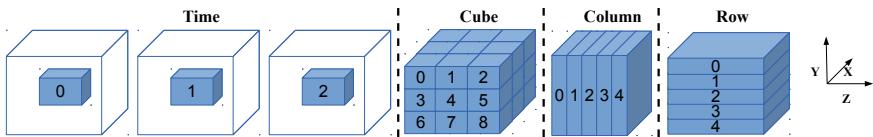


Fig. 5. Subvolume decompositions used in our evaluation (contiguous in order Z, Y, X , time). Blocks are labeled by accessing client.

the layouts used in the experiments. Additionally, we test each decomposition using independent I/O with all eight processes on each node participating. Note that a single participant-per-node configuration exhibited similar behavior on our test system, and replication on collective I/O access patterns (e.g., large, contiguous patterns) shows performance similar to our fully contiguous data decomposition, although with communication overhead.

Figure 6 shows performance under the different decompositions both before and after RADAR data replication. For these runs, we synchronize prior to running the decomposition and calculate bandwidth with respect to the maximum elapsed time for each individual read. We note a few points about these experiments:

1. All decompositions except the time-based decomposition decompose a data size of 2 GB (four timesteps of 512 MB volumes). Thus, an increasing number of clients leads to lower average request sizes and a higher number of requests, leading to potentially less efficient accesses when not using collective I/O. Additionally, access patterns for the non-time-based decompositions under a single client are fully contiguous, resulting in a performance regression between the original data layout and RADAR. We are currently unable to diagnose this difference; the generated layout by RADAR is the same, and the I/O driver follows largely the same code path.
2. The time-based decomposition defines a fixed-size subvolume for each client to read of total size 64 MB. As the number of clients increase, the per-client requests remain the same, leading to an increase in the total request size.
3. The cube decomposition divides the volume into perfect cubes (1, 8, 27, 81, 125, etc.) of no less than the number of clients, and clients are assigned multiple blocks to read, resulting in varying request granularities based on the number of clients. For instance, a four-client run will divide the subvolume into eight blocks and assign two blocks to each client. This approach can lead to both load imbalance (processes can be oversubscribed blocks compared with others) and varied access patterns because of the possibility of multiple smaller blocks combining into a single, large, contiguous block.

The time-based decomposition in Figure 6 shows low aggregate performance without replication due to numerous noncontiguous accesses. The reorganization through RADAR enables high performance across the spectrum, although performance tapers off once I/O servers begin processing requests from larger client counts.

The cube-based decomposition in Figure 6 results in block sizes of high variance with a changing number of clients, which is a significant factor in the overall

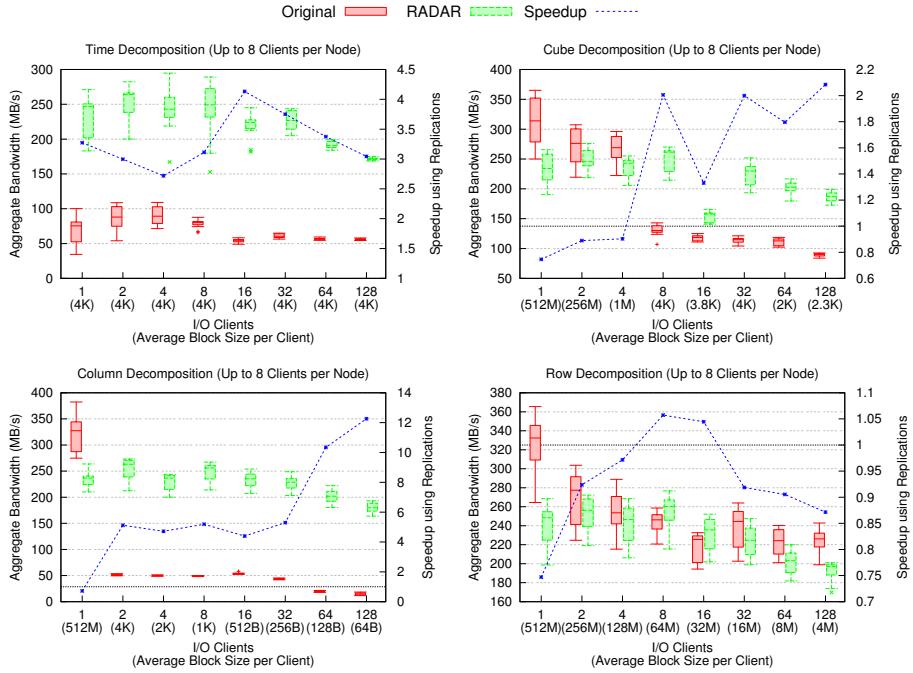


Fig. 6. Decomposition performance under the original data layout and with the RADAR-replicated layout

performance. In the figure, the performance implications can easily be seen between the four-client and eight-client decomposition for the nonreplication case, and the eight-client and 16-client decomposition for the RADAR case. Regardless, the use of RADAR helps smooth out the performance characteristics as a result of making the strided accesses contiguous per client and over/underprovisioning of replicas based on load.

The column-based decomposition in Figure 6 represents a pathological I/O pattern, as seen by the average contiguous block sizes. Hence, performance without collective optimizations or RADAR is far worse than any of the other decompositions as the number of clients increase. RADAR can greatly improve performance over the original data layout, although it also tapers off somewhat for increasing client counts.

The row-based decomposition in Figure 6 represents the “best case” for parallel I/O without reorganization: large, contiguous, non-overlapping blocks. Here, since the storage is the primary bottleneck and block sizes are very large, RADAR is not shown to have any benefit.

3.3 Model Verifications

We now look at how the performance modeling approach compares with the performance shown in Section 3.2. The goal of the performance models is to show whether a specific data layout can be improved by a modified data layout via replication. As

opposed to strict performance accuracy, the primary measurement of interest is the accuracy of relative performance between two layouts (one with replicas, one without), as our models do not perform high-fidelity simulation.

Figure 7 shows the results, comparing the model-derived performance of both the original layout and the layout under replication with the median of the performance shown in Section 3.2. We additionally show the estimated performance using the coarse-grained model corresponding to the best layout. In general, the “best layout” found corresponds to the heuristic of spreading each pattern’s data across as many servers as possible until overlap occurs, in which case the distribution contracts accordingly.

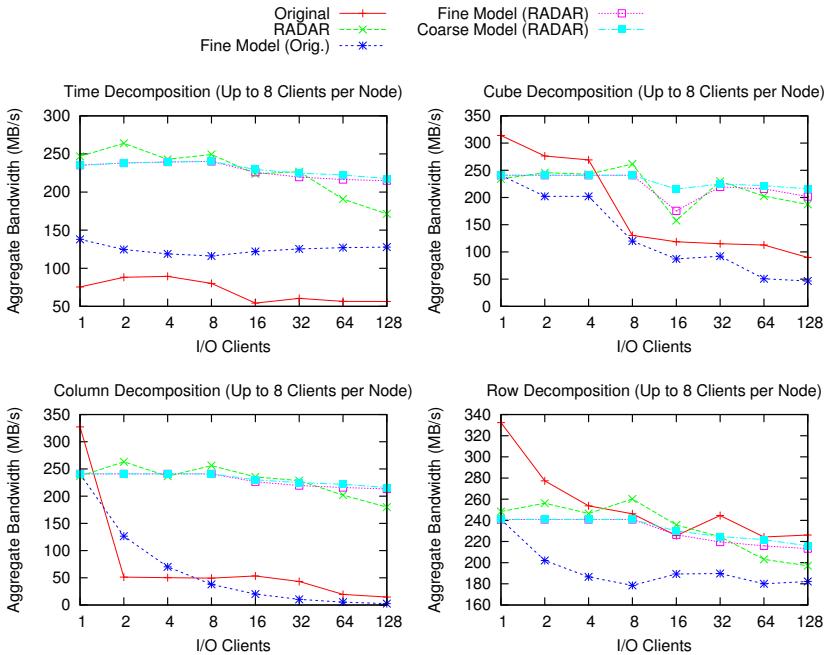


Fig. 7. Model results against median empirical performance (8 clients per node)

For the RADAR-controlled data layouts, the fine- and coarse-grained models follow the actual performance relatively closely, which is expected as RADAR produces contiguous and regular access patterns based on the model output. For the 16-client cube decomposition, the fine-grained model for the RADAR-generated layout correctly tracks the performance degradation due to the imbalance of per-client data requested, which the coarse-grained model misses.

For the original data layouts, the model results for the column and cube decompositions roughly follow the performance trend, although with somewhat less accuracy than for the RADAR-generated layouts. However, the model results still correctly predict RADAR as producing a performance benefit. The model of the time-based decomposition consistently overestimates performance, which we believe to be the result of

optimism in the readahead component of the fine-grained model. However, the model comparisons are still useful for determining that use of RADAR can result in significant performance gains. The model for the row decomposition, however, shows false positives due to underestimating the performance. Since the striping under the original layout corresponds to an all-client-to-all-server access pattern, we believe the underestimation is due to underestimating the ability of the system to handle concurrent requests, not to mention pipelining. Overall, we believe improving the resource contention and scheduling aspects of the model will reduce or eliminate these inconsistencies, although it must be performed in a computationally efficient manner to render it useful for optimization purposes.

4 Related Work

4.1 Replication in Storage, I/O Systems

Data replication in storage systems is a well-researched topic in many domains. Many parallel/distributed filesystems, such as the Hadoop Distributed File System [40] and the Google File System [11,29], built for task-centric, data-intensive workloads, as well as the Ceph filesystem [51], have data replication as a first-order feature. Local filesystem replication has also been explored in a performance context by reorganizing data to minimize rotational latency and maximize locality [2,17]. Additionally, hybrid methods are being explored in parallel filesystems without intrinsic replication support, such as a *shim* layer for PVFS allowing Hadoop-style workloads and replication [44].

Database systems also widely use replication, both for fault tolerance and as a performance optimizer. For instance, replicas can be created by using query history as a guide [31,52] or in a more dynamic approach where replication/indexing occurs as queries are performed [19,18].

The use of replication to ensure high availability and/or performance has also been explored through high-level libraries and I/O middleware. For MPI-based applications, works have shown that file block replication using the PMPI interface provides application-level I/O resiliency [41], while replicating data in different storage layouts can be used to improve performance for one-to-one, process-to-file configurations [42,56]. Additionally, specific metrics can be optimized by replication and reorganization, such as minimizing disk head thrashing by examining local disk traces with DiskSim [3,57].

4.2 Capturing and Detecting I/O Access Patterns

Many approaches have been developed to systematically derive system usage information from applications. For MPI-based applications, the MPI Parallel Environment (MPE) [30] provides full MPI event tracing, while mpiP [49] provides lightweight, statistical measures. The ScalaTrace family of MPI tracers focuses on compressed trace generation [32,36,50,54], using histogram generation and a combination of intranode and internode trace compression. Dynamic instrumentation methods include automatically instrumenting at compile time through source code analysis [24], as well as runtime binary instrumentation through IOPin [23], based on the Pin [27] framework.

For a “big-picture” view, Darshan [6,5] focuses on *centerwide* usage patterns by combining local, subsystem metrics (such as the Sysstat [12] and fsstats [9] utilities) and application-level metrics (instrumented through POSIX and MPI-IO).

Once acceptable profiles or logs of application/system performance are gathered, they can be mined for emergent patterns. Statistical learning methods can be used in a general sense to capture high-level patterns such as block-to-block association [28,33,48,2]. Recent methods have been developed specifically for HPC, again typically through the MPI/MPI-IO layers. Examples include strided pattern analysis for MPI prefetching [4] and pattern recognition for PLFS index compression under a checkpointing use-case [16]. The IOSig [4] trace analyzer converts I/O operations to compact and parameterized representations called I/O signatures using a *template matching* approach, which iteratively attempts to match specific patterns (e.g., regularly strided) to the sequence of I/O accesses.

5 Conclusion

Effective data distribution in large-scale analysis systems is an integral component of achieving high-performance I/O, especially in the presence of complex, noncontiguous workloads such as the volume decompositions we have presented. Through the tight coupling with a filesystem view of the data as a set of distinct objects, we were able to create arbitrary data layouts optimized for the access patterns induced on the dataset, all in a single container. RADAR is a promising step in the direction of automated specialization of data layouts based on application-specific needs and access patterns, providing both increased performance and an initial ability to reason about the “worth” of potential data layouts.

Acknowledgments. This work was supported by the U.S. Department of Energy, Office of Science, under Contract No. DE-AC02-06CH11357, as well as the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research and the U.S. National Science Foundation (Expeditions in Computing and EAGER programs).

References

1. Bent, J., Gibson, G., Grider, G., McClelland, B., Nowoczynski, P., Nunez, J., Polte, M., Wingate, M.: PLFS: A checkpoint filesystem for parallel applications. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC 2009, pp. 21:1–21:12. ACM, New York (2009)
2. Bhadkamkar, M., Guerra, J., Useche, L., Burnett, S., Liptak, J., Rangaswami, R., Hristidis, V.: BORG: Block-reORGanization for self-optimizing storage systems. In: Proceedings of the 7th Conference on File and Storage Technologies, FAST 2009, pp. 183–196. USENIX Association, Berkeley (2009)
3. Bucy, J.S., Schindler, J., Schlosser, S., Ganger, G.: Contributors. The DiskSim simulation environment version 4.0 reference manual. Technical Report CMU-PDL-08-101, Carnegie Mellon University Parallel Data Lab (2008)
4. Byna, S., Chen, Y., Sun, X.-H., Thakur, R., Gropp, W.: Parallel I/O prefetching using MPI file caching and I/O signatures. In: International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2008, pp. 1–12. IEEE (2008)

5. Carns, P., Harms, K., Allcock, W., Bacon, C., Lang, S., Latham, R., Ross, R.: Understanding and improving computational science storage access through continuous characterization. *ACM Transactions on Storage (TOC)* 7(3), 8:1–8:26 (2011)
6. Carns, P., Latham, R., Ross, R., Iskra, K., Lang, S., Riley, K.: 24/7 characterization of peta-scale I/O workloads. In: *IEEE International Conference on Cluster Computing, Cluster 2010*, pp. 1–10 (2009)
7. Carns, P.H., Ligon III, W.B., Ross, R.B., Thakur, R.: PVFS: A parallel file system for Linux clusters. In: *Proceedings of the 4th Annual Linux Showcase and Conference*, pp. 317–327 (2000)
8. Carns, P.H., Ligon III, W.B., Ross, R.B., Wyckoff, P.: BMI: A network abstraction layer for parallel I/O. In: *Workshop on Communication Architecture for Clusters, Proceedings of IPDPS 2005*, Denver, CO (April 2005)
9. Dayal, S.: Characterizing HEC storage systems at rest. Technical Report CMU-PDL-09-109, Carnegie Mellon University Parallel Data Laboratory (2008)
10. Frazier, M.W.: *An Introduction to Wavelets through Linear Algebra*. Springer (1999)
11. Ghemawat, S., Gobioff, H., Leung, S.-T.: The Google File System. In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, SOSP 2003*, pp. 29–43. ACM, New York (2003)
12. Godard, S.: Sysstat utilities home page,
<http://sebastien.godard.pagesperso-orange.fr/index.html>
13. Gong, Z., Boyuka II, D.A., Zou, X., Liu, Q., Podhorszki, N., Klasky, S., Ma, X., Samatova, N.F.: PARLO: Parallel Run-time Layout Optimization for scientific data explorations with heterogeneous access patterns. In: *13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2013)*, Delft, The Netherlands (2013)
14. Gong, Z., Rogers, T., Jenkins, J., Kolla, H., Ethier, S., Chen, J., Ross, R., Klasky, S., Samatova, N.F.: MLOC: Multi-level layout optimization framework for compressed scientific data exploration with heterogeneous access patterns. In: *Proceedings of the 41st International Conference on Parallel Processing, ICPP 2012* (2012)
15. Goodell, D., Kim, S.J., Latham, R., Kandemir, M., Ross, R.: An evolutionary path to object storage access. In: *Proceedings of the Seventh Workshop on Parallel Data Storage, PDSW 2012* (2012)
16. He, J., Bent, J., Torres, A., Grider, G., Gibson, G., Maltzahn, C., Sun, X.-H.: Discovering structure in unstructured I/O. In: *Proceedings of the Seventh Workshop on Parallel Data Storage, PDSW 2012* (2012)
17. Huang, H., Hung, W., Shin, K.G.: Fs2: Dynamic data replication in free disk space for improving disk performance and energy consumption. In: *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles, SOSP 2005*, pp. 263–276. ACM, New York (2005)
18. Idreos, S.: Database Cracking: Towards Auto-tuning Database Kernels. PhD thesis, University of Amsterdam (2010)
19. Idreos, S., Kersten, M., Manegold, S.: Database cracking. In: *Proceedings of the 3rd International Conference on Innovative Data Systems Research, CIDR 2007* (2007)
20. Interleaved or random (IOR) parallel filesystem I/O benchmark,
<https://github.com/chaos/ior>
21. Jenkins, J., et al.: ALACRITY: Analytics-driven lossless data compression for rapid in-situ indexing, storing, and querying. In: Hameurlain, A., Küng, J., Wagner, R., Liddle, S.W., Schewe, K.-D., Zhou, X. (eds.) *TLDKS X. LNCS*, vol. 8220, pp. 95–114. Springer, Heidelberg (2013)
22. Jenkins, J., Schendel, E., Lakshminarasimhan, S., Boyuka II, D.A., Rogers, T., Ethier, S., Ross, R., Klasky, S., Samatova, N.F.: Byte-precision level of detail processing for variable precision analytics. In: *ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*, Salt Lake City, UT, USA (2012)

23. Kim, S.J., Son, S.W., Liao, W.-K., Kandemir, M., Thakur, R., Choudhary, A.: IOPin: Runtime profiling of parallel I/O in HPC systems. In: 7th Parallel Data Storage Workshop, PDSW 2012 (2012)
24. Kim, S.J., Zhang, Y., Son, S.W., Prabhakar, R., Kandemir, M., Patrick, C., Liao, W.-k., Choudhary, A.: Automated tracing of I/O stack. In: Keller, R., Gabriel, E., Resch, M., Dongarra, J. (eds.) EuroMPI 2010. LNCS, vol. 6305, pp. 72–81. Springer, Heidelberg (2010)
25. Lakshminarasimhan, S., Jenkins, J., Arkatkar, I., Gong, Z., Kolla, H., Ku, S.-H., Ethier, S., Chen, J., Chang, C.S., Klasky, S., Latham, R., Ross, R., Samatova, N.F.: ISABELA-QA: query-driven analytics with ISABELA-compressed extreme-scale scientific data. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC, pp. 31:1–31:11. ACM, New York (2011)
26. Lawder, J.K., King, P.J.H.: Querying multi-dimensional data indexed using the Hilbert Space-Filling Curve. SIGMOD Record 30 (2001)
27. Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2005, pp. 190–200. ACM, New York (2005)
28. Madhyastha, T.M., Reed, D.A.: Learning to classify parallel input/output access patterns. IEEE Transactions on Parallel and Distributed Systems 13(8), 802–813 (2002)
29. McKusick, M.K., Quinlan, S.: GFS: Evolution on fast-forward. Queue 7(7), 10:10–10:20 (2009)
30. MPI parallel environment (MPE),
<http://www.mcs.anl.gov/research/projects/perfvis/software/MPE/>
31. Narayanan, S., Catalyurek, U., Kurc, T., Kumar, V.S., Saltz, J.: A runtime framework for partial replication and its application for on-demand data exploration. In: High Performance Computing Symposium, SCS Spring Simulation Multiconference, HPC 2005 (2005)
32. Noeth, M., Ratn, P., Mueller, F., Schulz, M., de Supinski, B.R.: ScalaTrace: Scalable compression and replay of communication traces for high-performance computing. Journal of Parallel and Distributed Computing 69(8), 696–710 (2009)
33. Oly, J., Reed, D.A.: Markov model prediction of I/O requests for scientific applications. In: Proceedings of the 16th International Conference on Supercomputing, ICS 2002, pp. 147–155. ACM, New York (2002)
34. Parallel I/O benchmarking consortium,
<http://www.mcs.anl.gov/research/projects/pio-benchmark/>
35. Pascucci, V., Frank, R.J.: Global static indexing for real-time exploration of very large regular grids. In: International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2001 (2001)
36. Ratn, P., Mueller, F., de Supinski, B.R., Schulz, M.: Preserving time in large-scale communication traces. In: Proceedings of the 22nd Annual International Conference on Supercomputing, ICS 2008, pp. 46–55. ACM, New York (2008)
37. Schmuck, F., Haskin, R.: GPFS: A shared-disk file system for large computing clusters. In: Proceedings of the 1st USENIX Conference on File and Storage Technologies, FAST 2002. USENIX Association, Berkeley (2002)
38. Schwan, P.: Lustre: Building a file system for 1000-node clusters. In: Proceedings of the 2003 Linux Symposium (2003)
39. Shorter, F.: Design and analysis of a performance evaluation standard for parallel file systems. Master's thesis, Clemson University (2003)
40. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The hadoop distributed file system. In: Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2010, pp. 1–10. IEEE Computer Society, Washington, DC (2010)
41. Son, S.W., Latham, R., Ross, R., Thakur, R.: Reliable MPI-IO through layout-aware replication. In: Proceedings of the 7th IEEE International Workshop on Storage Network Architecture and Parallel I/O, SNAPI 2011 (2011)

42. Song, H., Yin, Y., Chen, Y., Sun, X.-H.: A cost-intelligent application-specific data layout scheme for parallel file systems. In: Proceedings of the 20th International Symposium on High Performance Distributed Computing, HPDC 2011, pp. 37–48. ACM, New York (2011)
43. Song, H., Yin, Y., Sun, X.-H., Thakur, R., Lang, S.: A segment-level adaptive data layout scheme for improved load balance in parallel file systems. In: IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pp. 414–423 (2011)
44. Tantisiriroj, W., Son, S.W., Patil, S., Lang, S.J., Gibson, G., Ross, R.B.: On the duality of data-intensive file system design: Reconciling HDFS and PVFS. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2011, pp. 67:1–67:12. ACM, New York (2011)
45. Thakur, R., Choudhary, A.: An extended two-phase method for accessing sections of out-of-core arrays. *Scientific Programming* 5(4), 301–317 (1996)
46. Thakur, R., Gropp, W., Lusk, E.: An abstract-device interface for implementing portable parallel-I/O interfaces. In: Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation, FRONTIERS 1996, pp. 180–187. IEEE Computer Society, Washington, DC (1996)
47. Thakur, R., Ross, R., Lust, E., Gropp, W.: Users guide for ROMIO: A high-performance, portable MPI-IO implementation. Technical Report ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory (2004)
48. Tran, N., Reed, D.A.: Automatic ARIMA time series modeling for adaptive I/O prefetching. *IEEE Transactions on Parallel and Distributed Systems* 15(4), 362–377 (2004)
49. Vetter, J.S., McCracken, M.O.: Statistical scalability analysis of communication operations in distributed applications. In: Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming, PPoPP 2001, pp. 123–132. ACM, New York (2001)
50. Vijayakumar, K., Mueller, F., Ma, X., Roth, P.C.: Scalable I/O tracing and analysis. In: Proceedings of the 4th Annual Workshop on Petascale Data Storage, PDSW 2009, pp. 26–31. ACM, New York (2009)
51. Weil, S.A., Brandt, S.A., Miller, E.L., Long, D.D.E., Maltzahn, C.: Ceph: A scalable, high-performance distributed file system. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation, OSDI 2006, pp. 307–320. USENIX Association, Berkeley (2006)
52. Weng, L., Catalyurek, U., Kurc, T., Agrawal, G., Saltz, J.: Servicing range queries on multidimensional datasets with partial replicas. In: IEEE International Symposium on Cluster Computing and the Grid, CCGrid 2005, vol. 2, pp. 726–733. IEEE (2005)
53. Witten, I.H., Moffat, A., Bell, T.C.: Managing Gigabytes: Compressing and Indexing Documents and Images, 2nd edn. Morgan Kaufmann (1999)
54. Wu, X., Vijayakumar, K., Mueller, F., Ma, X., Roth, P.C.: Probabilistic communication and I/O tracing with deterministic replay at scale. In: Proceedings of the 2011 International Conference on Parallel Processing, ICPP 2011, pp. 196–205. IEEE Computer Society, Washington, DC (2011)
55. Yin, Y., Byna, S., Song, H., Sun, X.-H., Thakur, R.: Boosting application-specific parallel I/O optimization using IOSIG. In: Cluster, Cloud and Grid Computing (CCGrid), pp. 196–203 (2012)
56. Yin, Y., Li, J., He, J., Sun, X.-H., Thakur, R.: Pattern-direct and layout-aware replication scheme for parallel i/o systems. In: IEEE International Symposium on Parallel and Distributed Computing, IPDPS 2013, pp. 345–356 (2013)
57. Zhang, X., Jiang, S.: InterferenceRemoval: Removing interference of disk access for mpi programs through data replication. In: Proceedings of the 24th ACM International Conference on Supercomputing, ICS 2010, pp. 223–232. ACM, New York (2010)
58. Zobel, J., Moffat, A.: Inverted files for text search engines. *ACM Computing Surveys* 38(2) (July 2006)

Fast Multiresolution Reads of Massive Simulation Datasets

Sidharth Kumar¹, Cameron Christensen¹, John A. Schmidt¹,
Peer-Timo Bremer^{1,4}, Eric Brugger⁴, Venkatram Vishwanath², Philip Carns²,
Hemanth Kolla³, Ray Grout⁵, Jacqueline Chen³, Martin Berzins¹,
Giorgio Scorzelli¹, and Valerio Pascucci¹

¹ SCI Institute, University of Utah, Salt Lake City, UT, USA

² Argonne National Laboratory, Argonne, IL, USA

³ Sandia National Laboratory, Livermore, CA, USA

⁴ Lawrence-Livermore National Laboratory, Livermore, CA, USA

⁵ National Renewable Energy Laboratory, Golden, CO, USA

`sidharth.kumar@utah.edu`

Abstract. Today’s massively parallel simulation codes can produce output ranging up to many terabytes of data. Utilizing this data to support scientific inquiry requires analysis and visualization, yet the sheer size of the data makes it cumbersome or impossible to read without computational resources similar to the original simulation. We identify two broad classes of problems for reading data and present effective solutions for both. The first class of data reads depends on user requirements and available resources. Tasks such as visualization and user-guided analysis may be accomplished using only a subset of variables with a restricted spatial extent at a reduced resolution. The other class of reads requires full resolution multivariate data to be loaded, for example to restart a simulation. We show that utilizing the hierarchical multiresolution IDX data format enables scalable and efficient serial and parallel read access on a variety of hardware from supercomputers down to portable devices. We demonstrate interactive view-dependent visualization and analysis of massive scientific datasets using low-power commodity hardware, and we compare read performance with other parallel file formats for both full and partial resolution data.

Keywords: parallel I/O, multiresolution, PIDX, read performance, interactive visualization, S3D, Uintah, VisIt.

1 Introduction

Massively parallel scientific simulations often generate large datasets that can range in size up to many terabytes. Reading this data is required for several reasons. Due to crashes, code modifications or limited job time, simulations must often be restarted from an intermediate timestep. Restarts require reading of the entire saved state of the simulation at full resolution for a given time. Similar to simulation restarts, comprehensive postprocessing analysis also requires reading

of full resolution data, although only a subset of variables may be necessary. The other major reasons to read data are for user-directed analysis and visualization. Unlike restarts and postprocessing, these tasks can often be performed using lower-resolution data with less spatial extent. For example, coarse resolution data can be used to compute an approximation of comprehensive analysis results. Similarly, the limited size of display devices permits lower-resolution data to be shown with no perceptible difference in visual quality. Finally, spatial extent can be restricted since only data within a visible region actually needs to be loaded. Thus, we identify two general classes of data reading for large scientific data: (a) full resolution reads of an entire dataset as required for simulation restarts, and (b) partial resolution reads of a subset of the data suitable for visualization and cursory analysis tasks. Complete reads are generally performed using parallel systems while partial reads may be done in serial or parallel.

We describe three improvements for reading massively parallel simulation checkpoint data at both full and partial resolutions. First, we demonstrate the utility of coarse resolution reads using spatially restricted subsets of the domain for view-dependent rendering in which only data within the field of view at the granularity necessary to support the display device needs to be loaded. Spatially restricted coarse resolution reads facilitate faster data loading as well as the ability to visualize massive simulation data using modest commodity hardware on which it would have previously been impossible to load full datasets. We provide a plugin and infrastructural modifications to enable view-dependent rendering using the VisIt visualization tool, a distributed parallel application currently in use on many supercomputers. Next, we demonstrate fast full resolution file loads of complete checkpoint data for parallel simulation restarts or comprehensive analysis. Finally, we evaluate parallel reading of data at varying resolutions suitable for summary analysis and visualization. Our work includes three significant contributions for reading massive simulation data:

- interactive view-dependent visualization of massive datasets using VisIt
- efficient parallel reads of complete checkpoint data for simulation restarts
- parallel multiresolution reads for summary analysis

In order to address both parallel full resolution reads, as well as parallel or serial partial resolution reads we utilize the IDX data format [20]. IDX is a hierarchical multiresolution data format with support for both lightweight serial and fast distributed parallel I/O. Parallel I/O is performed using PIDX[15], a fast parallel library for reading and writing IDX multiresolution data.

The remaining sections of this paper are organized as follows. Section 2 explains relevant background work involving the IDX multiresolution file format, view-dependent rendering and the VisIt visualization and analysis application. Section 3 explains the setup of our experiments and the integration of the PIDX I/O library with the Uintah simulation code. Sections 4-6 discuss our experimental results with view-dependent serial reads, full resolution parallel reads and multiresolution parallel reads. We discuss related work in Section 7 and conclude in Section 8.

2 Background

One of the important observations of our work is that utilizing a hierarchical multiresolution data format facilitates faster and less expensive visualization with no apparent reduction in quality. For our experiments we utilize the IDX format. We briefly introduce this format below as well as the PIDX library for parallel I/O. Next, we discuss the mechanism of view-dependent rendering. Finally, we describe the VisIt parallel distributed visualization system for which we have integrated support for reading IDX data.

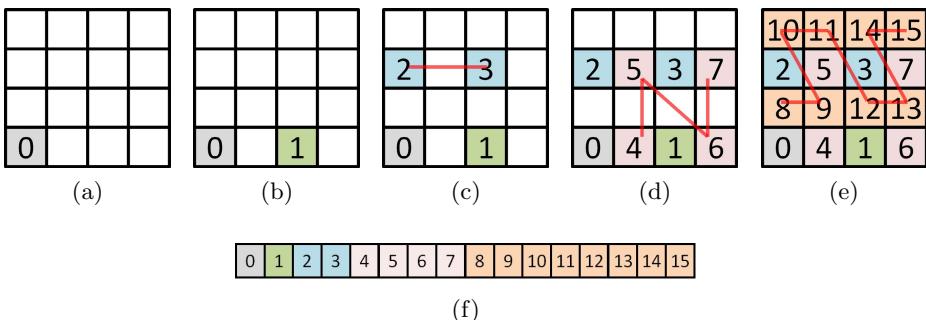


Fig. 1. HZ ordering (indicated by box labels and curves) at various levels for a 4x4 dataset. (a)-(e) Levels 0-4, respectively. (f) File level arrangement of data points in the IDX format.

2.1 IDX Multiresolution File Format

IDX is a multiresolution file format that enables fast and efficient access to large-scale scientific data. The format provides efficient, cache oblivious, progressive access to data by utilizing a hierarchical Z (HZ) order for storage [20]. The HZ order is calculated for each data sample using the spatial coordinates of that sample. The samples are then assigned a particular HZ level, analogous to the different levels of resolution at which data can be retrieved, from coarse to fine (see figure 1(a-e)). With each increasing level, the number of elements increases by a factor of 2. Data in an IDX file is written with an increasing HZ order (see figure 1(f)).

The data access layer of IDX format has three salient features that make it particularly attractive. First, the order of the data is independent of the out-of-core block structure, so that its use in different settings (e.g., local disk access or transmission over a network) does not require any data reorganization. This property is especially important in a parallel setting to facilitate restarts with varying numbers of nodes and reading data that originally might have been produced by a different number of nodes. Second, conversion from the Z-order indexing [16] used in classical database approaches to the IDX HZ-order indexing

scheme can be implemented with a simple sequence of bit-string manipulations. Finally, since there is no redundant data stored for different resolutions, IDX does not incur the increased storage requirements associated with most other hierarchical and out-of-core data management schemes.

2.2 PIDX: Parallel IDX

IDX is a desirable file format for visualization of large-scale simulation results because of its ability to access multiple levels of resolution with low latency for interactive exploration. The Parallel IDX (PIDX) library enables parallel simulations to directly write IDX data [13]. The library coordinates data access among processes to concurrently write to the same dataset with coherent results.

Our previous work has been dedicated to the design and optimization of PIDX for writing data in the IDX format, as opposed to this work that focuses on reads. In [13], we demonstrated the use of PIDX in coordinating parallel write access to multidimensional, regular datasets, and explored several low-level encoding and file access optimizations. In subsequent work [15], we introduced a two-phase I/O strategy [9] in which each process first performs a local HZ encoding of its own data. The data is then aggregated by a subset of processes (aggregators) that in turn write it to disk. This strategy avoids suboptimal small-sized file accesses from each process in lieu of large, contiguous accesses performed by a select few aggregators.

2.3 View-Dependent Rendering

View-dependent rendering is a combination of techniques designed to facilitate interactive graphical applications consisting of frustum-based scene culling and level-of-detail based geometry and texture loading. Scene culling, commonly known as “frustum clipping,” is an integral part of the OpenGL and DirectX rasterization pipelines [3] and most types of interactive applications from games to CAD systems rely on this technique for fast drawing [2]. The notion is simple: data outside the viewing frustum need not be considered for shading a given scene. One example of level-of-detail texture loading is MIP mapping, introduced by [27], which automatically determines the optimal resolution texture data to be used when rendering a scene from a given viewpoint based on the distance of the texture to the viewer. These maps are typically precomputed from full-resolution textures and stored on disk as an image “pyramid” where each successive level is half the size of the previous level, effectively doubling the size of the original image. MIP mapping has two major advantages: (a) it avoids loading unnecessary texture data into limited graphics memory, and (b) it permits faster rasterization since textures need not be sampled more than necessary. However, for extremely large textures or 3D volumes, computing and storing these image pyramids is generally too cumbersome.

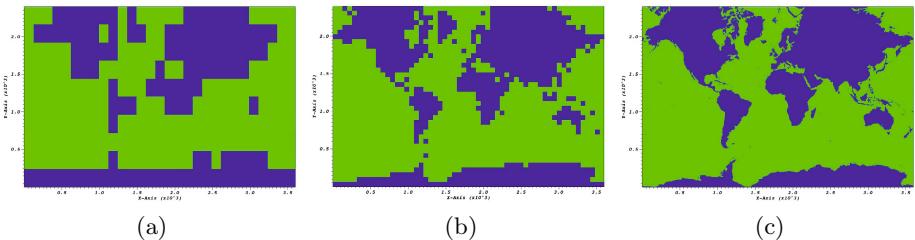


Fig. 2. Coarse-to-fine resolution (HZ) levels of a 2D dataset shown in VisIt

2.4 VisIt Visualization Application

VisIt [5] is a popular distributed parallel visualization and analysis application used by many scientists. VisIt is typically executed in parallel where it coordinates visualization and analysis tasks for massive simulation data. The data is typically read at its full resolution, requiring as much system memory as the size of the data being read to perform its work. In order to leverage the multiresolution capability of the IDX format, we have added a plugin to read IDX data and incorporated dynamic view-dependent read capabilities with VisIt as well as enabling visualization of IDX data hosted on remote servers. With these additions it is now possible to perform view-dependent visualization and analysis using a tiny fraction of the compute power previously required for these tasks. Manual control of the resolution level is also permitted. Figure 2 shows an IDX dataset in VisIt at different resolution levels.

2.5 Related Work

Parallel I/O file formats have been developed to aid in structuring data to facilitate efficient data storage and access. Some of the popular parallel file formats are Parallel HDF5 [1], Parallel NetCDF [17], MPI-IO [24], VAPOR [4] and ADIOS [18].

Multiresolution data formats for parallel processing environments have been studied previously in [11,25,6]. Chiueh leverages Gaussian and Laplacian Pyramid transforms tailored according to the underlying storage architecture to improve read performance [8]. Chaoli, Jinzhu and Han have presented work in parallel visualization [26] of multiresolution data. Their algorithm involves conversion of raw data to a multiresolution wavelet tree and focuses more on parallel visualization rather than a generic data format such as PIDX. More recently, DynaM data representation supports convolution-based multiresolution data representation [25]. Ahrens et. al. work enables multiresolution visualization by sampling existing data, and writes the multiresolution levels and meta-data to disk as independent files while keeping the full-resolution file intact [6].

The ViSUS Viewer [22,21] is an application that facilitates online visualization and analysis. It has been designed to allow interactive exploration of massive

scientific models on a variety of hardware from desktops down to hand-held devices. ViSUS supports thread-parallel operation in contrast to the distributed parallel mechanism provided by VisIt. The ViSUS Viewer natively supports the IDX streaming data format.

3 Experiment Setup

The experiments presented in this work were performed on Edison at the National Energy Research Scientific Computing (NERSC) Center and Tukey at the Argonne Leadership Computing Facility (ALCF). Edison is a Cray XC30 with a peak performance of 2.39 petaflops, 124,608 compute cores, 332 TiB of RAM and 7.5 PiB of online disk storage. We used Edison Lustre file system (168 GiB/s, 24 I/O servers and 4 Object Storage Targets). Tukey is a visualization cluster consisting of 96 compute nodes. Each node consists of two 2GHz AMD Opteron 6128 8 core CPUs (16 cores per node and 64 GB RAM) and two NVIDIA Tesla M2070 GPUs (6 GB GPU RAM). All nodes are connected via QDR Infiniband interconnect and share the same GPFS filesystem with Mira (a 768K core Blue-Gene/Q system). We used one of the /project scratch filesystems for doing the visualization read experiments.

S3D Simulation Code: S3D is a continuum scale first principles direct numerical simulation code that solves the compressible governing equations of mass continuity, momenta, energy and mass fractions of chemical species including chemical reactions. The computational approach in S3D is described in [7]. Each run of S3D generates four variables: pressure, temperature, velocity (3 samples) and species (11 samples). Details of integration of the PIDX I/O library with S3D are described in [15].

Uintah Software Framework: Uintah is a general purpose software framework used in the development of components for the numerical modeling of simulations involving fluid-structure interactions, computational fluid dynamics, solid mechanics and multi-physics simulation on core counts approaching 768K cores[19].

Uintah's native data output format, Uintah Data Archive (UDA), consists of individual timesteps (directories) each containing binary data and meta-data files written per MPI rank. The UDA format was originally designed in the mid 90s when large simulations were on the order of 2K cores. Several thousand files per directory were considered manageable. However, with core counts approaching 1 million cores, data output consisting of hundreds of thousands of small files is untenable from both a file system/OS and scalable I/O point of view. We have incorporated the PIDX library into the Uintah framework as a optional, high throughput, scalable I/O system for managing data output and checkpointing.

4 View-Dependent Multiresolution Serial Reads

It is often desirable to utilize low-cost, low-power hardware to visualize massive simulation data as well as to perform cursory or user-directed analysis. For

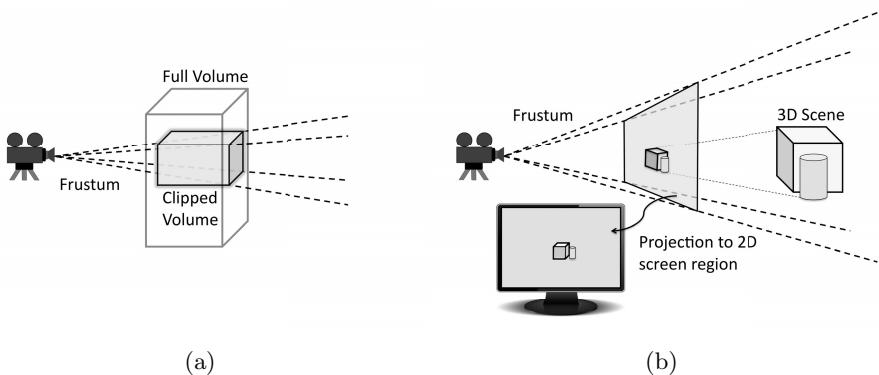


Fig. 3. Example images showing the mechanism of view-dependent data reading. (a) Shows how the spatial extent of the data is clipped by the camera viewing frustum. Only data within the frustum must be loaded. (b) Shows the projection of a 3D scene onto the 2D camera viewing plane, which is copied to the display. The ratio of the area of a screen pixel to the projected area of a 3D volume unit is used to determine the resolution (HZ) level that will be loaded. For example, if 4 volume units project to one pixel, 1/4 resolution data is sufficient to convey the scene.

example, a portable device could be used to monitor the progress of a simulation. Utilizing a hierarchical multiresolution data format enables view-dependent determination of the spatial extent and resolution of data to be loaded, permitting less data to be requested and facilitating interactive navigation of arbitrarily large datasets. In this section we will explain the mechanism of view-dependent data loading and the implementation of this technique in the VisIt visualization application. We compare read performance in a variety of scenarios.

4.1 View-Dependent Data Loading

All visualization systems incorporate some type of 2D or 3D camera model in order to transform data into a 2D image on a screen. The camera model can be distilled down to a matrix multiplication applied to the individual components of the data in order for them to appear in the desired location when projected onto the 2D viewing plane. A comprehensive description of the viewing pipeline is outside the scope of this work, but the process is explained in most introductory computer graphics texts such as [23,12,10]. The matrix resulting from the composition of model, viewing and projection transformations, as well as the near and far clipping planes, creates a viewing “frustum”, a parallelepiped oriented in the world space coordinate system. The frustum is used to crop data outside the view of the virtual camera (called “clipping”) and the ratio of the projected size of a data unit compared to a screen pixel can be used to determine the optimal resolution of data to be loaded.

View-dependent data loading begins by applying the frustum clip planes to the volume of data being visualized. Clipping the bounds of the data volume with the

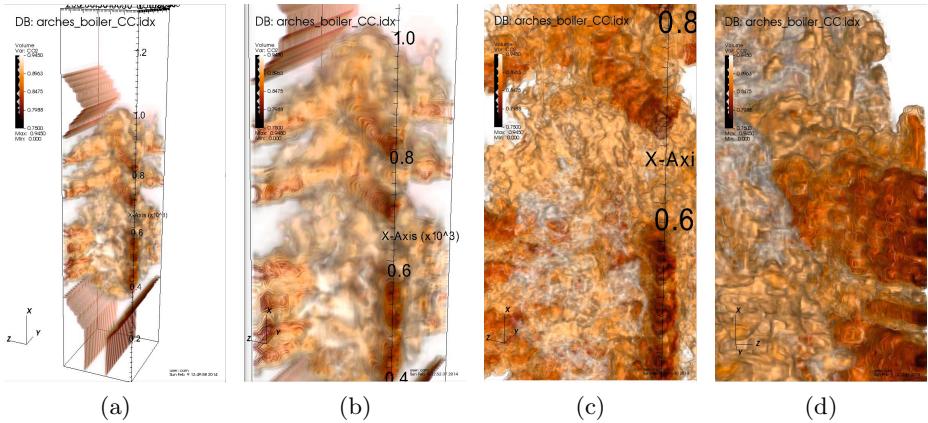


Fig. 4. Images showing view-dependent data loading for a Uintah combustion simulation. From left to right the view is continually refreshed while the camera is zoomed closer to the data.

viewing frustum is how we determine the spatial extent of the data to be loaded. See figure 3(a). Next we use the relative size of a display pixel compared to the size of a unit of the data volume projected to the screen in order to determine the minimum resolution necessary to visualize the data without artifacts. See figure 3(b). This technique is similar to the ubiquitous MIP-mapping technique of [27] used to decide what level of a given texture should be sampled based on its projection to screen space. Using the extent of the volume and relying on the fact that every HZ level reduces the resolution by half in one direction, we can determine the minimum level necessary for a unit of the data volume to cover approximately one pixel on the screen. Loading higher resolution data than this is a waste of time and memory because multiple data units are simply averaged over the same pixel, and loading a lower resolution will result in an overly coarse or “jaggy” image.

The combination of these two techniques allows for interactive exploration of any size data using even modest hardware resources. We describe the mechanism by which we have incorporated view-dependent rendering into the VisIt visualization system (see figure 4).

4.2 Implementation in VisIt

We have incorporated the IDX format as both a serial and parallel database reader plugin for VisIt.

Loading an entire large data volume is frequently excessive for simple visualization or cursory analysis and requires significant compute resources. For summary analysis and visualization, or to load extremely large datasets using more modest hardware, a coarse approximation of the data can be sufficient. Our implementation of the serial IDX reader in VisIt facilitates loading coarse levels

of the data by selecting a sub-volume and resolution sufficient to accommodate a given viewing frustum. Additionally, utilizing the IDX format with VisIt enables a new capability for remote visualization: loading remotely hosted volumes is now possible by simply specifying the URL to be loaded. The remote server that provides IDX data has no knowledge of VisIt nor is there a need for VisIt to be installed at the remote location.

4.3 Performance Evaluation

Interactivity is the overall performance goal of using view-dependent data loading. The key idea is to load as little data as possible at one time while still providing sufficient coverage of the viewing region. Efficient data loading is an important component of engineering an interactive application, but additional support is required at every level. For example, it is important to frequently check for user input even during a complicated analysis or visualization operation or while reading data.

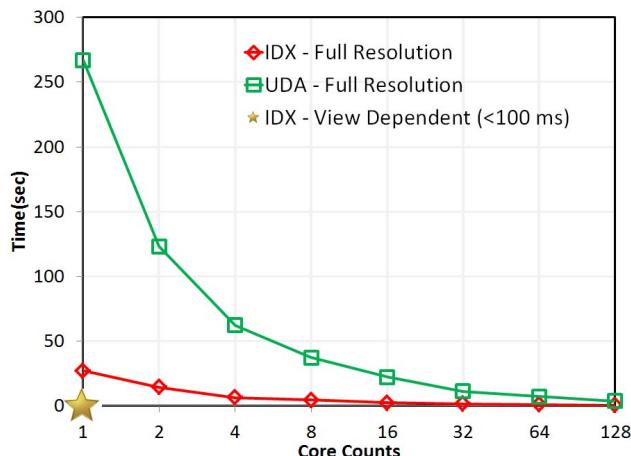


Fig. 5. VisIt load times on Tukey for a Uintah Data Archive (UDA) file versus the same simulation stored in the IDX format for increasing numbers of cores. The star in the bottom left indicates the time for loading the data using the serial view-dependent method.

Figure 5 shows the results of our experiments on Tukey. The times required by VisIt for loading a Uintah Data Archive (UDA) file versus the same simulation stored in the IDX format, are plotted using increasing numbers of cores. In order to compare view-dependent loading of multiresolution data to the existing practice of loading full resolution data using a visualization cluster, the point noted with a star at the bottom of the figure shows the average time for loading the data using serial view-dependent visualization. All load times were gathered using the '-timings' argument to VisIt. The dataset shown was computed using 9920 processors and consists of 19 variables on a $1323 \times 335 \times 290$ domain over 171 timesteps totaling approximately 4 TiB. The data shown in the figure is for loading a single variable of the dataset. Note that the differences observed in

loading UDA vs IDX data at full resolution are due to the nature of the two file formats. IDX data is stored in a cache-oblivious layout, whereas data in UDA data is stored in naïve row-major order using one file per process. Because the UDA format uses so many files, additional access time (open/close/seek) is incurred compared to the IDX format, which uses fewer files to store the same data. UDA is comparable to one file per process I/O, so as more cores are used the overhead incurred in accessing files is reduced because the number of files that are accessed per process decreases. Most importantly, as can be seen in the chart, view-dependent data reading is significantly faster than loading full resolution data using any number of cores.

5 Parallel Reads at Full Resolution

Until now we have demonstrated the utility of multiresolution reads in serial. However, full-resolution parallel reads are still important for simulation restarts and comprehensive postprocessing analysis. In this section we describe the read implementation of the PIDX library followed by its in-depth evaluation showing both strong and weak scaling results. We also compare PIDX performance with two common distributed parallel file formats, MPI collective I/O [24] and one file per process S3D I/O.

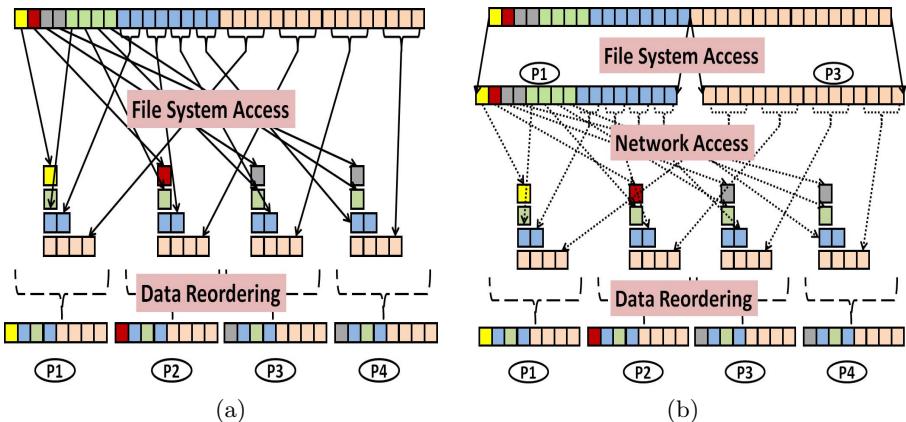


Fig. 6. Reading IDX: data is read into a hierarchy of z-buffers and then re-ordered to the conventional multidimensional application layout. (a) The one-phase approach that requires many small non-contiguous disk accesses to load data into the z-buffer hierarchy of each process. (b) The improved two-phase method. First, large contiguous blocks of data are read from disk by a few aggregators. Next, the data is loaded via the network into the z-buffer hierarchies of each process.

5.1 Read Implementation

Parallel IDX reads involve translation from the hierarchical progressive data order of the IDX format to the conventional multidimensional data layout suitable for use by application processes. We use our experience with the PIDX writer to design the corresponding reader [13]. We first describe a naïve implementation that uses one I/O phase to fetch data. Next, we improve upon this strategy by adding an additional data aggregation phase to achieve scalable performance by reducing noncontiguous data access.

For the one-phase parallel reader, every process first reads the data at each HZ level (recall figure 1 from section 2.1), and then correctly orders the data for application usage. This strategy is complementary to the approach followed in one-phase writes, where every process first calculates the HZ order and corresponding HZ level for its sub-volume, and then uses independent MPI-I/O write operations to store each level. For both reading and writing this method entails a high degree of noncontiguous data access of the file system, dramatically impeding scalability [15]. See figure 6a.

To improve upon the naïve one-phase approach, we devised a two-phase I/O algorithm to mitigate the problems caused by large numbers of small-sized disk accesses. With this approach, only a select few processes assigned as aggregators access the file system, making large-sized contiguous read requests. Once the data is read by the aggregators, it is next transferred to every process over the network for re-ordering. As with PIDX writes, one-sided MPI communication is used for this purpose: after the aggregators finish reading data from the file system, all the processes use MPI_get() to gather data directly from the aggregators' remote buffers. Figure 6b illustrates the two-phase I/O approach.

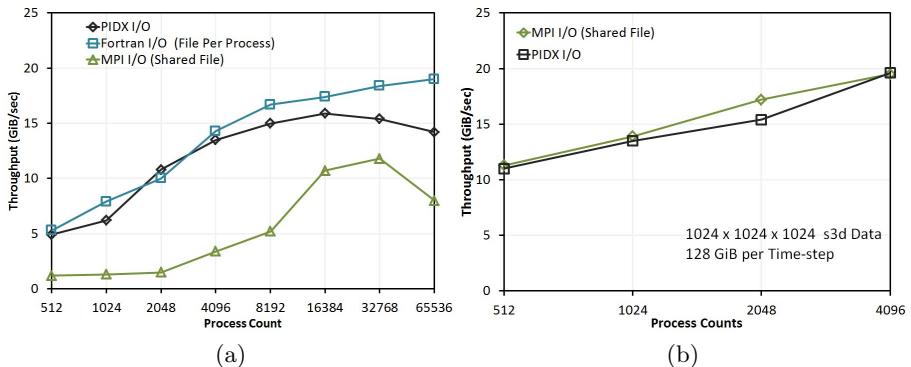


Fig. 7. [S3D I/O Benchmark] results using Edison (a) weak scaling of different I/O mechanisms including PIDX, Fortran I/O, and MPI I/O reading datasets with a block size of 32^3 . (b) Strong scaling for PIDX and MPI I/O for reading datasets of dimension 1024^3 .

5.2 Performance Evaluation

We perform two sets of experiments: weak scaling and strong scaling. Weak scaling increases the data size proportional to the number of processors. It is a useful measurement for determining the efficiency of reading data for simulation restarts that typically require reading a complete dataset using many cores. Strong scaling increases the number of processors while maintaining the same problem size. It is a useful metric to determine how much a particular task can be accelerated by the addition of computational resources. Strong scaling is an indicator of parallel visualization and analysis performance because these tasks are often performed using fewer cores than the original simulation.

Weak Scaling (simulation restarts): We use the S3D I/O simulator to benchmark the performance of parallel reads. We compared PIDX performance with that of both the Fortran I/O and MPI collective I/O modules in S3D. With Fortran I/O, data is present in its native format organized into files equal to the number of processes used to create them. PIDX determines the number of files based on the size of the dataset rather than on the number of processes [15]. In the case of MPI I/O, data exists in a single, shared file. As opposed to Fortran I/O, both PIDX I/O and MPI I/O have an extensive data communication layer as part of the collective I/O phase. Default file system striping parameters were used for Fortran I/O, whereas for MPI I/O and PIDX I/O the Lustre striping was increased to span all 96 OSTs available on that system.

In order to benchmark read performance we first used the S3D simulator to create datasets of the corresponding format at each process count. We then invoked the S3D postprocessing module to enable parallel reads and conducted our experiments. For each run, S3D reads data corresponding to 25 timesteps. Every process reads a 32^3 block of double-precision data (4 MiB) consisting of four variables: pressure, temperature, velocity (3 samples) and species (11 samples). We varied the number of processes from 512 to 65,536, thus varying the amount of data read per timestep from 4 MiB to 256 GiB.

Considering the performance results in figure 7a, we observe that PIDX scales well up to 32,768 processes, and that for all process counts, it performs better than MPI I/O. We also observe that both PIDX and MPI I/O do not perform as well compared to Fortran I/O because unlike PIDX and MPI I/O, Fortran I/O does not require any data exchanges across the network. The performance pattern of PIDX is mainly due to its aggregation phase, which finds a middle ground between the single-shared-file MPI I/O and one file per process I/O.

One new observation that we make for parallel I/O is the decline in scalability for both MPI I/O and PIDX at higher core counts, whereas Fortran I/O shows continued scaling. This decline is in contrast to parallel writes for which continued positive scaling results have been demonstrated for all I/O formats [14]. One possible explanation for this behavior is that network communication involved in the collective I/O data scattering phase may play a role in this effect. If this were the case, we note that Fortran I/O would not be affected since each process is independent of the network. We believe the poor scaling of data reads

will become increasingly problematic in the future and that further investigation may be worthwhile.

Strong Scaling (Post-processing analysis and visualization): Using the S3D I/O simulator, we compared the strong scaling results of PIDX with MPI I/O for 25 timesteps on Edison. We used the S3D simulator to create input datasets of the corresponding format using 32768 cores with a per process domain size of 32^3 . We then invoked the S3D postprocessing module to enable parallel reads and conducted our experiments. For each run, S3D reads data corresponding to 25 timesteps. Because the S3D simulator does not make available the capability to read multiple files per process, we were unable to acquire strong scaling numbers for Fortran I/O. Our experiments used a 3D domain of dimension 1024^3 . We varied the number of processes from 512 to 4096, for which the block size per process ranged from 128^3 (512 processes) down to 64^3 (4096 processes). The results of this study are shown in Figure 7b. We observe that PIDX and MPI I/O perform similarly at all core counts.

6 Parallel Reads at Varying Resolution

For our final experiments we explore parallel reading of multiresolution data. We already discussed serial multiresolution reads in Section 4, and parallel full resolution reads in Section 5. Providing parallel multiresolution data loading confers the advantages of faster coarse resolution data access while allowing for greater processing power and total aggregate memory available on larger clusters.

Parallel multiresolution reads can be useful for visualization and postprocessing analysis. As with serial multiresolution data reading, relatively small compute clusters can be utilized to perform analysis or to visualize much larger datasets than was previously possible. In addition, dedicated visualization clusters such as Tukey may have associated GPU arrays or be connected to large displays such as those used for power walls or immersive caves. These clusters may be utilized to produce ultra-high-resolution imagery using rendering techniques such as ray-tracing or to perform postprocessing analysis using a mixture of GPU and CPU resources.

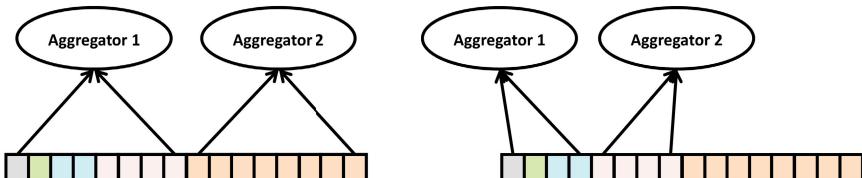


Fig. 8. Examples showing aggregation balancing for parallel multiresolution reads: (a) aggregator assignment when reading full-resolution data, and (b) aggregator assignment when reading half resolution data

6.1 Design

One of the major challenges for performing multiresolution reads in parallel is ensuring stable scalable performance by limiting data reads. We made two significant changes to extend parallel full resolution reads to support multiresolution capabilities. First, based on the desired level of resolution we added the capability to directly select the appropriate HZ levels. Second, we made the aggregation phase self-balancing by maintaining the number of aggregators for varying levels of resolution. See figure 8. By balancing the load across aggregators, we ensure a uniform reduction of workload.

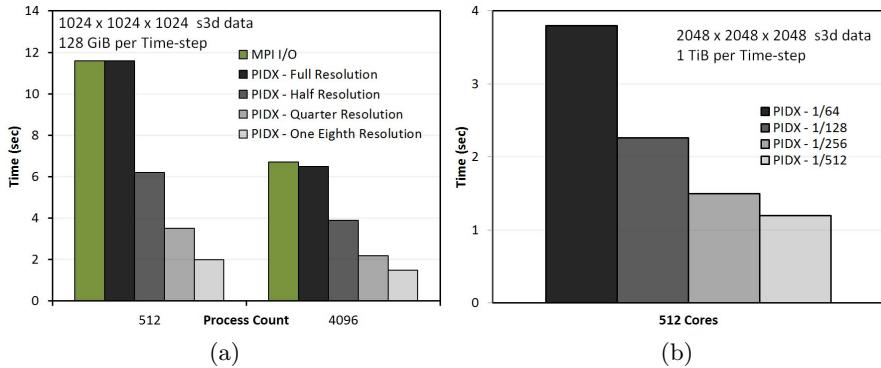


Fig. 9. [S3D I/O Benchmark] results using Edison. (a) Timings for reading full resolution 1024^3 volume data using PIDX and MPI I/O as well as timing for partial resolution reads using PIDX. (b) Timings for reading a 2048^3 volume data at partial resolutions using PIDX, simulating an environment in which it is not possible or desirable to read the full resolution data.

6.2 Performance Evaluation

Due to limitations in aggregate memory, loading data at full resolution would not be feasible for some datasets, but with multiresolution support, data can be loaded at coarser resolutions, enabling approximate analysis or visualization for extremely large datasets.

We conducted these experiments using Edison, with stripe setting similar to our parallel full resolution experiments. We first used the S3D simulator to create input datasets of global resolution 1024^3 and 2048^3 . For the 1024^3 volume we evaluate the performance of multiresolution reads by first reading the data at full resolution followed by reading the data at partial resolution. We decrease the resolution by half for each successive test down to $1/8$ of the original volume size. For comparison, we also show the corresponding result for loading the full resolution volume using MPI I/O (see figure 9(a)). In accordance with our strong scaling results, we observe that reading the full resolution data requires

approximately the same amount of time for both PIDX and MPI I/O. For multiresolution reads we see almost perfect scaling when the resolution is reduced by half, but as the resolution continues to decrease the efficiency begins to deteriorate due to the reduced workload of individual aggregators as the requested data size decreases.

We use the 1 TiB per timestep 2048^3 volume to simulate the situation in which the amount of data to load might be larger than the aggregate system memory. For this particular case we begin reading data at 1/64 resolution and decrease by half for each run down to 1/512. The results for this experiment can be seen in figure 9(b). From the figure we observe that the read time using 512 cores is less than four seconds, which is fast enough to be used for cursory visualization or to compute approximate analysis on an otherwise unwieldy dataset. Also, as observed with the 1024^3 case, we notice a nearly perfect scaling for the first decrease in resolution while successive reductions are less optimal.

7 Conclusion

We identified two broad classes of data reading: full resolution reads for simulation restarts or postprocessing analysis and partial resolution reads of spatial subsets suitable for cursory analysis and visualization. We have shown that using a hierarchical multiresolution data format enables scalable and efficient serial and parallel read access. The technique of view-dependent reading was incorporated into the VisIt visualization application to enable interactive visualization and analysis of massive datasets with very modest hardware resources. Two different application codes, S3D and Uintah, were used to compare native data formats with the IDX hierarchical multiresolution data format. We demonstrated that reading using PIDX weak scales well to 32K cores and is approximately 25% slower than Fortran I/O and approximately 50% faster than MPI I/O. We noted a falloff in scaling performance of reads for both PIDX and MPI I/O beyond 32K cores. We also demonstrated the strong scaling characteristic of PIDX. Finally, we described the use of parallel multiresolution reads to support faster visualization and approximation of comprehensive analysis when dealing with very large datasets.

Acknowledgements. This research used resources of the National Energy Research Scientific Computing Center and the Argonne Leadership Computing Facility at Argonne National Laboratory, supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357 and DE-AC02-05CH11231, respectively. It is based upon work supported by the Department of Energy, National Nuclear Security Administration, the Department of Energy Scalable Data Management, Analysis, and Visualization (SDAV) SciDAC Institute and PISTON, award numbers DE-NA0002375, DE-SC0007446, and DE-SC0010498, respectively.

References

1. HDF5 home page, <http://www.hdfgroup.org/HDF5/>
2. OpenGL standard, <http://www.opengl.org/>
3. OpenGL view frustum culling,
<http://www.lighthouse3d.com/tutorials/view-frustum-culling/>
4. VAPOR home page, <http://www.vapor.ucar.edu/>
5. Visit home page, <https://wci.llnl.gov/codes/visit/>
6. Ahrens, J.P., Woodring, J., DeMarle, D.E., Patchett, J., Maltrud, M.: Interactive remote large-scale data visualization via prioritized multi-resolution streaming. In: Proceedings of the 2009 Workshop on Ultrascale Visualization, UltraVis 2009, pp. 1–10. ACM, New York (2009)
7. Chen, J.H., Choudhary, A., de Supinski, B., DeVries, M., Hawkes, E.R., Klasky, S., Liao, W.K., Ma, K.L., Crummey, J.M., Podhorszki, N., Sankaran, R., Shende, S., Yoo, C.S.: Terascale direct numerical simulations of turbulent combustion using s3d. In: Computational Science and Discovery, vol. 2 (January 2009)
8. Chiueh, T.-C., Katz, R.H.: Multi-resolution video representation for parallel disk arrays. In: Proceedings of the First ACM International Conference on Multimedia, MULTIMEDIA 1993, pp. 401–409. ACM, New York (1993)
9. del Rosario, J.M., Bordawekar, R., Choudhary, A.: Improved parallel I/O via a two-phase run-time access strategy. SIGARCH Comput. Archit. News 21, 31–38 (1993)
10. Foley, J.D., van Dam, A., Feiner, S.K., Hughes, J.F.: Computer Graphics: Principles and Practice, 2nd edn. Addison-Wesley Longman Publishing Co., Inc., Boston (1990)
11. Guthe, S., Wand, M., Gonser, J., Strasser, W.: Interactive rendering of large volume data sets. In: Proceedings of the Conference on Visualization 2002, VIS 2002, pp. 53–60. IEEE Computer Society, Washington, DC (2002)
12. Hearn, D., Baker, M.P.: Computer graphics, C version, vol. 2. Prentice Hall, Upper Saddle River (1997)
13. Kumar, S., Pascucci, V., Vishwanath, V., Carns, P., Hereld, M., Latham, R., Peterka, T., Papka, M., Ross, R.: Towards parallel access of multi-dimensional, multi-resolution scientific data. In: 2010 5th Petascale Data Storage Workshop (PDSW), pp. 1–5 (2010)
14. Kumar, S., Vishwanath, V., Carns, P., Levine, J.A., Latham, R., Scorzelli, G., Kolla, H., Grout, R., Ross, R., Papka, M.E., Chen, J., Pascucci, V.: Efficient data restructuring and aggregation for I/O acceleration in pidx. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012, pp. 50:1–50:11. IEEE Computer Society Press, Los Alamitos (2012)
15. Kumar, S., Vishwanath, V., Carns, P., Summa, B., Scorzelli, G., Pascucci, V., Ross, R., Chen, J., Kolla, H., Grout, R.: Pidx: Efficient parallel I/O for multi-resolution multi-dimensional scientific datasets. In: Proceedings of the 2011 IEEE International Conference on Cluster Computing, CLUSTER 2011, pp. 103–111. IEEE Computer Society, Washington, DC (2011)
16. Lawder, J.K., King, P.J.H.: Using space-filling curves for multi-dimensional indexing. In: Jeffery, K., Lings, B. (eds.) BNCOD 2000. LNCS, vol. 1832, p. 20. Springer, Heidelberg (2000)

17. Li, J., Liao, W.-K., Choudhary, A., Ross, R., Thakur, R., Gropp, W., Latham, R., Siegel, A., Gallagher, B., Zingale, M.: Parallel netCDF: A high-performance scientific I/O interface. In: Proceedings of SC 2003: High Performance Networking and Computing, Phoenix, AZ. IEEE Computer Society Press (November 2003)
18. Lofstead, J., Klasky, S., Schwan, K., Podhorszki, N., Jin, C.: Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). In: Proceedings of the 6th International Workshop on Challenges of Large Applications in Distributed Environments, CLADE 2008, pp. 15–24. ACM, New York (2008)
19. Meng, Q., Humphrey, A., Schmidt, J., Berzins, M.: Investigating applications portability with the uintah dag-based runtime system on petascale supercomputers. In: Proceedings of the 2013 ACM/IEEE Conference on Supercomputing (SC 2013). ACM (2013)
20. Pascucci, V., Frank, R.J.: Global static indexing for real-time exploration of very large regular grids. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (2001)
21. Pascucci, V., Scorzelli, G., Summa, B., Bremer, P.-T., Gyulassy, A., Christensen, C., Kumar, S.: Scalable visualization and interactive analysis using massive data streams. Advances in Parallel Computing: Cloud Computing and Big Data 23, 212–230 (2013)
22. Pascucci, V., Scorzelli, G., Summa, B., Bremer, P.-T., Gyulassy, A., Christensen, C., Philip, S., Kumar, S.: The visus visualization framework. In: Bethel, E.W., Childs, H., Hansen, C. (eds.) High Performance Visualization: Enabling Extreme-Scale Scientific Insight, ch. 19, pp. 401–414. Chapman & Hall and CRC Computational Science (2012)
23. Shirley, P., Marschner, S.: Fundamentals of Computer Graphics, 3rd edn. A. K. Peters, Ltd., Natick (2009)
24. Thakur, R., Gropp, W., Lusk, E.: On implementing MPI-IO portably and with high performance. In: Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems, pp. 23–32. ACM Press (1999)
25. Tian, Y., Klasky, S., Yu, W., Wang, B., Abbasi, H., Podhorszki, N., Grout, R.: Dynam: Dynamic multiresolution data representation for large-scale scientific analysis. In: 2013 IEEE Eighth International Conference on Networking, Architecture and Storage (NAS), pp. 115–124. IEEE (2013)
26. Wang, C., Gao, J., Li, L., Shen, H.-W.: A multiresolution volume rendering framework for large-scale time-varying data visualization. In: Fourth International Workshop on Volume Graphics, pp. 11–223 (June 2005)
27. Williams, L.: Pyramidal parametrics. SIGGRAPH Comput. Graph. 17(3), 1–11 (1983)

Rebasing I/O for Scientific Computing: Leveraging Storage Class Memory in an IBM BlueGene/Q Supercomputer

Felix Schürmann¹, Fabien Delalondre¹, Pramod S. Kumbhar¹,
John Biddiscombe², Miguel Gila², Davide Tacchella², Alessandro Curioni³,
Bernard Metzler³, Peter Morjan⁴, Joachim Fenkes⁴, Michele M. Franceschini⁵,
Robert S. Germain⁵, Lars Schneidenbach⁵,
T.J. Christopher Ward⁶, and Blake G. Fitch⁵

¹ Blue Brain Project, Brain Mind Institute,
École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland

{felix.schuermann,fabien.delalondre,pramod.kumbhar}@epfl.ch
² CSCS, Swiss National Supercomputing Centre, Lugano, Switzerland

{biddisco,miguel.gila,tack}@cscs.ch

³ IBM Research GmbH, 8803 Rueschlikon, Switzerland

{cur,bmt}@zurich.ibm.com

⁴ IBM Deutschland Research & Development GmbH, 71032 Böblingen, Germany
{peter.morjan,fenkes}@de.ibm.com

⁵ IBM T.J. Watson Research Center Yorktown Heights, NY 10598, USA
{franceschini,rgermain,schneidenbach,bgf}@us.ibm.com

⁶ IBM Software Group, Hursley Park, Hursley SO212JN, U.K.
tjcw@uk.ibm.com

Abstract. Storage class memory is receiving increasing attention for use in HPC systems for the acceleration of intensive IO operations. We report a particular instance using SLC FLASH memory integrated with an IBM BlueGene/Q supercomputer at scale (Blue Gene Active Storage, BGAS). We describe two principle modes of operation of the non-volatile memory: 1) block device; 2) direct storage access (DSA). The block device layer, built on the DSA layer, provides compatibility with IO layers common to existing HPC IO systems (POSIX, MPI, HDF5) and is expected to provide high performance in bandwidth critical use cases. The novel DSA strategy enables a low-overhead, byte addressable, asynchronous, kernel by-pass access method for very high user space IOPs in multithreaded application environments. Here, we expose DSA through HDF5 using a custom file driver. Benchmark results for the different modes are presented and scale-out to full system size showcases the capabilities of this technology.

Keywords: data-intensive supercomputing, IO, storage class memory, IOPS, verbs.

1 Introduction

Despite the ever present ranking of supercomputers by their computational performance, there is a growing awareness of the need to include other system characteristics relevant for many scientific applications such as overall memory footprint, memory bandwidth and latency as well as access to storage[1–3]. This process is fueled by several developments: recent years of technology development yielded a scaling of logic circuits that was not matched by equivalent scaling of either capacity or bandwidth of the primary memory technology, DRAM. For scientific applications that do not share the property of high arithmetic intensity of standard benchmarks [4] this results in a diminishing fraction of utilized peak performance of those systems. Secondly, novel memory technologies are emerging, which exhibit properties distinctly different from classical DRAM or hard disk drives and present alternative design points in terms of density and energy efficiency as well as latency and bandwidth [5], hinting at opportunities to overhaul classical memory and IO system hierarchies. The term Storage-Class-Memory (SCM) has been coined to refer to these technologies which are typically non-volatile. Lastly, novel applications from the field of data analytics or data-intensive modeling are entering the HPC realm and their memory and IO requirements are driving interest in more cost-effective design points than conventional DRAM-only supercomputers fulfilling those requirements can offer.

Detailed bio-physical modeling of brain tissue, such as that pioneered by the Blue Brain Project [6–8], is an example of a complex scientific application hoping to exploit the properties of storage-class-memory in supercomputers in various scenarios and is the driver for this present study. One of the features of detailed brain modeling is the intrinsic heterogeneity of the brain: each neuron and each synapse is distinct, which in a computational representation leads to parameters and state variables unique to every modeled entity [9]. Combined with the large numbers of those entities, e.g. an estimated 200 million neurons and 10^{12} synapses for an entire rat brain [10], the resulting memory footprint is large and at the same time the algorithmic intensity low. With the human brain being an additional three orders of magnitude more complex, cellular models of the human brain will occupy a daunting estimated 100PB of memory that will need to be revisited by the solver at every time step.

Apart from the absolute scale and bandwidth challenges of detailed brain tissue models, advocating a paradigm of data-centric computing, there are numerous other, more mundane, aspects of today’s computational workflows that make the exploration of SCM interesting and progressively tractable: on the one hand, multiple steps are involved for building the computational representation of the brain tissue model [11], solving the time evolution of the equations by the simulator [12], and validating the structural and functional aspects [8]. On the other, analysis of those brain tissue models often requires transformation of the data representation. A classical example is that a visualization may require a portion of a model selected by view-frustum and spatial coordinates while the simulator utilizes an object-based decomposition [13]. Acceleration of large-scale, block-oriented access

as well as facilitation of concurrent small random reads limited by the number of IOPS will be beneficial for these workflows.

2 Related Work

Substantial work has been published on characterizing IO patterns of scientific applications and benchmarking classical disk-based file systems, see e.g. [14–19]. The complexity of the storage hierarchies and the number of parameters that can affect performance led to the development of high-level IO libraries such as ADIOS [20] or SIONlib [21] but tuning for performance is an ongoing challenge[22]. For the last several years the usage of flash in HPC applications has been proposed and investigated, mostly in form of using commodity SSDs and PCIe cards with flash[23–26, 1, 3].

A more systems-level view has been proposed in the Active Storage work [27, 28] that now includes development of the hardware/software stack from storage containers to middleware and also envisions the inclusion of storage-class-memory technologies beyond flash. Other system implementations comprising clusters of power-efficient nodes with flash storage have been explored at CMU [29, 30]. Another related activity is the RAMCloud work that is exploring the aggregation of large amounts of DRAM on a network to support data intensive applications that require very low latency access [31]. Considerations of where to integrate SCM continue [32]. The distinguishing feature of the Active Storage work is the combination of activities that include: 1) focusing on making storage-class-memories available as an aggregated global resource in a scale-out system 2) exploring the potential for parallel computational offload into the Active Storage layer (using general purpose CPUs and embedded processing elements) 3) research efforts at many levels in the hardware/software stack, particularly the exploration of novel methods for enabling access to storage-class-memories.

3 Research HPC System

A Blue Gene Active Storage (BGAS) system is defined by extending the standard IBM BlueGene/Q (BGQ) architecture [33] by the integration of storage-class-memory and an external network interface into each node. BGAS systems can be used in a stand-alone mode or as the I/O subsystem of a standard BlueGene/Q compute fabric. An instance of this, Blue Brain IV, has been built and installed by the Swiss National Computing Center (CSCS) in Lugano for and on behalf of the Blue Brain Project at EPFL in Lausanne.

This system is based on 4 standard racks of BGQ integrated with two types of components, which can be switched via software per partition of the BGQ: 1) The Blue Brain IV *standard configuration* connects the BGQ compute racks to standard IBM BlueGene/Q I/O Nodes via a proprietary network [34], which in turn are connected with 5 IBM GSS running GPFS [35] via QDR Infiniband. The nominal specifications of the system are: 4096 nodes of A2 compute chip [36] for a total of 0.8 Petaflops theoretical peak, 16 GB DRAM/node for a total

of 64 TB DRAM memory, standard IBM BlueGene/Q Compute Node Kernel. The attached storage is a 5 IBM GSS model 26 with 4.2 PB usable raw storage running GPFS v3.5. 2) In the *active storage configuration*, the BGQ compute racks are connected to a separate set of IONs, each augmented with a prototype PCIe card. This Hybrid Scalable Solid State Storage (HS4) card is a PCIe 2.0 x8 full height half length device that integrates 1.9 TB of SLC flash, DRAM and two 10 GbE ports. Via the 10GbE ports, these IONs are also connected to the IBM GSS system. The BG/Q IO drawer is a standard rack mount 3U, air cooled, chassis containing eight Blue Gene/Q nodes on a planar which enables a $2 \times 2 \times 2$ 3D torus among them. The IO torus is a research feature and enables Blue Gene/Q torus connectivity among large numbers of I/O nodes by extending the torus with optical cables among IO drawers. The Blue Brain IV system has a 64 node BGAS system cabled as a $4 \times 4 \times 4$ node 3D torus with a total of about 120TB SLC flash.

The BGAS software stack extends the RHEL 6 based Blue Gene/Q IO node software environment [37] providing a general purpose multi-tenancy operating system capable of hosting both storage and application functions. The BGAS extensions are integrated into a single image, which is network loaded on all BGAS nodes by the Blue Gene/Q control system during partition boot.

4 Direct Storage-Class-Memory Access (DSA)

The integration of SCM presents significant challenges to the classic I/O stack. Like flash, emerging memory technologies can have limited endurance requiring wear-leveling and other management. They may read and write with asymmetric latency and bandwidth characteristics and may require explicit erase operations. In general, new memories seem to be trending toward more complex characteristics which will be best utilized in hybrid devices where the memories serve complementary roles. Additionally, the processors accessing these hybrid memory devices are likely to have an increasingly large core count.

In order to address those challenges, as well as the specific challenge of flash memory latencies, a storage interface is required that includes enabling many deep queues of asynchronous work requests. To deliver the full capability of the SCM to applications, such an interface should be directly accessible from user space, by-passing the significant overheads of the classic block I/O stack. Finally, in order to avoid another I/O interface change when truly byte addressable SCM becomes cost-effective, we set that requirement for the current interface. To this end, a novel SCM interface based on the Open Fabrics Enterprise Distribution (OFEDTM) [38] Remote Direct Memory Access (RDMA) interface has been defined: the “Direct SCM Access” (DSA) method. DSA enables the RDMA-style memory registration of SCM regions as well as user application virtual memory regions and enables zero-copy transfers among them. The registration of storage is analogous to the registration of virtual memory for OFED RDMA.

DSA is the primary interface for the BGAS HS4 device. The DSA driver is implemented as an OFED RDMA verbs provider and plugs into the standard

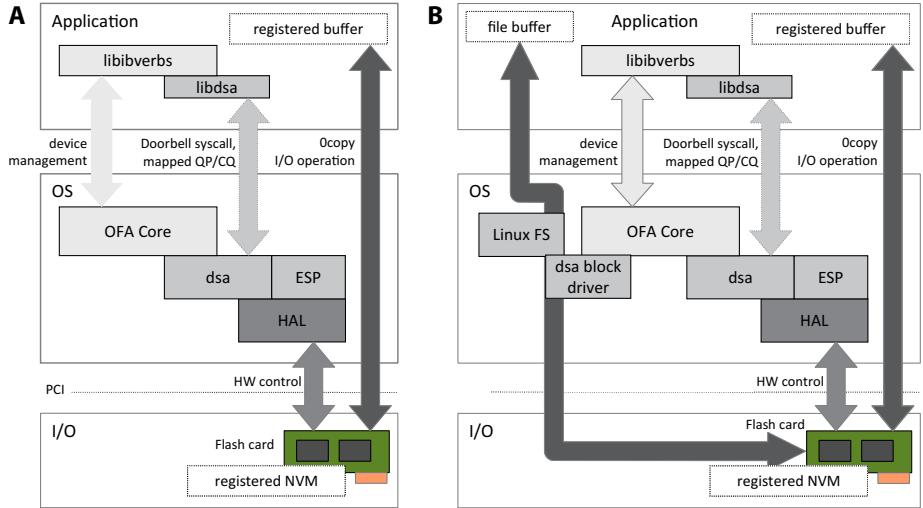


Fig. 1. A) Direct SCM Access (DSA) module implemented as a regular RDMA verbs provider that plugs into the OFED framework. B) Adding support for block based NVM access to DSA.

OFED RDMA framework. The DSA driver includes a module called an “Embedded Storage Peer” (ESP) and this is the only process which can be connected to by the DSA driver. Once a DSA user process connects to the ESP, OFED RDMA send/receive type messages are used to open and close storage memory partitions in a request/response manner. DSA uses the two sided RDMA communication model (Send and Receive) for control operations on the HS4 device, and the one-sided RDMA Reads and Writes to read and write flash content.

Figure 1A depicts a high level design overview of the components and their interactions with the OpenFabrics core as well as with the HS4 hardware. As with any OpenFabrics RDMA provider, DSA comprises two main software modules: 1) A loadable kernel module which includes the ESP, attaches to the OFA core, and further interacts with the hardware abstraction layer (HAL). DSA establishes a fast data path for flash read and write operations from user level. This fast data path is implemented as application private work and completion queues represented by memory regions shared between DSA and the user level library. 2) A dynamically linkable user library, libDSA, which links with the generic libibverbs OFA library to provide both DSA specific RDMA control path and fast path. Interacting with the DSA kernel module via the generic OFA control path, libDSA manages communication contexts and queue pair/completion queues.

Figure 1B shows the integration of HS4 memory resources into the Linux file system. The DSA block driver module bridges between legacy block-based memory access and the byte addressable nature of the DSA API. The DSA kernel module exports a kernel level RDMA verbs interface accessed by the DSA block driver via the OpenFabrics kernel RDMA verbs interface. The DSA block driver

in turn provides a generic block driver interface used by the Linux file system services. This also offers opportunities to exploit the features of HS4 card and build journaling file systems that make use of the other available SCM types.

4.1 Exposing DSA Using HDF5

In order to expose DSA level access directly to applications, we have extended a previously developed Virtual File Driver (VFD) for the HDF5 Parallel IO library. A full description of the driver with examples of usage is provided in [39, 40]. The driver maps a region of Distributed Shared Memory (DSM) hosted on nodes within a parallel job and makes them visible to the HDF5 library as a byte addressable file space hosted in RAM instead of disk. This allows for high speed access to/from a file in memory which can be shared by coupled applications allowing them to exchange information using the HDF5 API. A natural extension of this is to map the file addresses to flash memory distributed over the HS4 cards and thereby create a shared file with persistence so that not only can data be shared between coupled applications, but the persistent nature of the memory allows it to be reused between sessions.

The DSM driver exposes RAM directly to the MPI layer to enable one sided transfers to take place to/from memory on a hosting node to the application. Since the DSA presents a view of flash memory that is behind an API, it is no longer possible to perform one sided put/get operations directly to the user application and it is necessary to add an additional virtualization layer between DSM address and the storage. Data reads/writes from flash are therefore locally buffered on each node and then passed to MPI for exchange to remote nodes. This has the disadvantage of reducing performance relative to the RAM based approach (in addition to the higher latency), but does allow for arbitrary sized HDF5 files to be created as one is no longer limited by the available/addressable RAM on each DSM node and may create a virtual space as large as the combined sizes of the flash partitions available.

5 Benchmarks

dsa_bench is a micro-benchmark created to test and measure the DSA stack performance from user space. It is configurable via command line parameters and can access all partitions and types of memory on the HS4 hardware. The benchmark posts the specified number of requests until the queue limit is reached. Then it polls for a completion before posting the next request. The command line arguments also allow controlling the alignment of addresses in host memory and in HS4 storage. This is especially useful for testing memories other than flash or walking alternative code paths for flash access because flash can only be written in 8 KiB increments. IOPS are measured using settings with deep queues and fewer signalled requests with small transfer sizes (e.g. 8 KiB, with 512 outstanding requests and a signal every 32 requests). Bandwidth measurements are done with at least 64 1 MiB requests.

By default, the benchmark performs 2 phases, an initial sequential write over a given range of memory on the HS4 card and a second phase, by default comprising random reads, over the same memory range. A separate transfer data size can be specified for each phase. The second phase can be configured to use random or sequential access for pure reads and writes or a 50:50 mix of reads and writes. Data verification and different data patterns are possible too. The user controls the depth of the queues and the number of un signalled requests between signalled requests. Un signalled requests complete without completion queue entries and are completed with the next signalled request. Therefore, un signalled requests are posted in batches together with a signalled request as the last in a batch.

The Linux utility **dd** is being used to measure bandwidth for sequential reads/writes from/to flash configured as block device.

IOR is a highly configurable MPI application designed to measure parallel file system read/write I/O performance using various interfaces and access patterns. The version of the code used for this work is available on GitHub [41] master branch with aa604c1d38de803aa0db3f3abb5515e0ed1857ea SHA1. IOR executables have been compiled with both POSIX and MPI-I/O interfaces using MPICH2 (V1.5) [42] and XLC (V12.0) or gcc (V4.4.7) compilers on IBM Blue Gene/Q or BGAS cluster respectively. GPFS file systems created on the flash of the HS4 cards for this study use 16 MiB block and 4 GiB page pool size (per ION) for bandwidth measurements and 128 KiB block and 1 GiB page pool size (per ION) for IOPS measurements; for bandwidth measurements, IOR BlockSize (**\$IORBlockSize**) was set to 160 GiB on BGAS and 10 GiB on BGQ CNK and respectively to 32 GiB on BGAS and BGQ CNK for IOPS measurements. Accordingly, the page pool sizes are always well below 1 % of the transferred data to avoid caching effects. Execution of IOR benchmark directly on BGAS or from Blue Gene/Q compute nodes to BGAS nodes allowed measuring bandwidth and IOPS performance. IOPS using IOR are computed from the bandwidth for 4 KiB IOR data transfer size, where the file opening time is negligible. Depending on the benchmark the number of MPI processes/node (**\$MPIproc**) was varied. IOR was invoked with the following parameters: `ior -b $IOR_BlockSize -r10 -p $MPIproc -t $TransferBlockSize -a POSIX/MPIIO -u -I`.

HDF5 bench is a simple program using the HDF5 API to open a file collectively across a series of nodes and writes a 1D hyperslab of data per process into the file, such that the hyperslabs cover the entire file space. The flash-DSM size is set to a multiple of the number of processes used with each process writing a (configurable) fixed size chosen in this case to be 256 MiB each, with an average timing from 15 runs being taken. To smooth out network contention, the DSM is set to use Block Cyclic memory mapping, with a block size of 4 MiB, implying that each 4 MiB chunk of data maps to a consecutive MPI rank of the DSM. This implies that each DSA put operation (IOP) is also 4 MiB in this example.

6 Early Benchmark Results and Discussions

6.1 Micro-benchmarks

The BGAS stack can be measured at several standard and novel interfaces to evaluate the cost/benefit of each layer. In Table 1a, we present a snapshot of BGAS stack micro-benchmark results for flash measured at the DSA layer and at the verbs block device layer. Table 1b contains small block size IOPS results for on-card DRAM measured at the DSA layer to provide some indication of the performance possible with memory technologies other than flash. Benchmark results on the IBM Power7+ system serve as a baseline reference for the capability of the HS4 device.

Table 1. Single card read (R) and write (W) micro-benchmarks for both Blue Gene/Q and Power 7+ platforms. All read benchmarks used a random access pattern while all write benchmarks used sequential access. We observed relatively small performance differences due to random versus sequential access. The bandwidth data were acquired using 1 MB transfer block sizes while the IOPS results are reported for a selection of small block sizes. Flash access performance is reported for single-threaded tests as well as the best performance achieved (Number of threads in parentheses or “==” if single-threaded performance was the best). Results for 32 Byte random IOPS achieved to DRAM on the HS4 card are provided to showcase the low overheads possible using the DSA software stack.

Flash Performance:			BG/Q		P7+	
			Single Thr.	Best Conf.	Single Thr.	Best Conf.
DSA client	BW	W(1 MB) (MB/s)	1100 1240	== 2270 (2)	2710 3020	== ==
	IOPS	W(8 KB) R(8 KB) R(4 KB)	65k 75k 85k	91k (4) 200k (3) 180k (3)	270k 360k 600k	== == 700k (2)
	Lat.	W(8 KB) (μs)	490 133	== ==	440 78	== ==
	VBD	BW (dd)	W(1 MB) (MB/s)	860 2100	1000 (2) 2200 (2)	1300 3000
					2200 (2)	
						==

(a) Results for accessing flash storage via DSA directly or via the verbs block device (VBD).

32 Byte DRAM Access			
	BG/Q	P7+	Thr.
W	120k	710k	1
W	230k		2
W	340k		3
R	120k	740k	1
R	235k	1060k	2
R	340k	1050k	3

(b) IOPS performance results for 32 Byte accesses to DRAM storage on the HS4 card.

The DSA bandwidth results show both Blue Gene/Q and P7+ approaching the PCIe 2.0 x8 limit (after encoding overheads) of 3.2 GB/s for flash reads. We note that two processes are required on Blue Gene/Q to achieve best performance on flash reads while P7+ requires just a single thread. On P7+ we achieve high write bandwidth using a single thread however, on Blue Gene/Q, the performance results indicate that more processes will be required to achieve

full bandwidth. Similarly, DSA achieves our target of 200k IOPs using 3 processes on Blue Gene/Q while P7+ sets a baseline of 600k IOPs on one thread. We have implemented the Verbs Block Device (VBD) to enable a standard Linux Block Device interface using DSA to enable files systems such as Ext4 and GPFS to utilize storage-class-memory. The overhead associated with layering a block device is shown in Table 1a.

In Table 1b we report on DSA performance while accessing DRAM. We make all the memories on the HS4 card available via DSA and benchmarking DRAM enables us to explore the performance limits of our stack independent of flash attributes. Read and write IOPs scale well with increased threads on Blue Gene/Q while Power7+ requires fewer processors for similar performance. Micro-benchmarking DSA accesses to DRAM gives a snapshot of the current limits of the software stack on a given platform. We exceed our target of 200k IOPs on Blue Gene/Q with three processes and achieve over one million IOPs on Power7+. These results show that the BGAS stack built around DSA will exceed our performance targets on power efficient, many core architectures.

6.2 Block Device Interface

The block device interface represents the easiest path for an application to adopt the BGAS capabilities since no modifications are necessary if standard IO libraries such as POSIX and MPIO are used. In order to map out what an application may expect from BGAS in this mode, we measured the bandwidth (IOPS) as function on transfer size for a GPFS with settings as described in Section 5 for the two different scenarios. Figure 2 shows the respective GPFS performance of a single BGAS node. Single node performance is the average performance per node on an 8 ION GPFS partition where $\$MPIproc = 1$. In line with the micro-benchmarks, read performance is about twice the write performance for large transfers in terms of bandwidth and about 50% larger in terms of small IOPS. The different IO interfaces (POSIX file per process/shared file; MPIO file per process/shared file) at this scale are fairly equivalent.

As also indicated by the micro-benchmarks, bandwidth performance for block device operations on the HS4 card shows some but not a very large dependency on the number of threads used of the A2 processor. If more MPI processes per node are used for the IOR benchmark, maximal read and write performance is reached at smaller transfer sizes (Figure 3; left panel) and maximal read performance is slightly improved. It should be noted that the overhead introduced by GPFS for larger reads and writes is about 20-30% when compared to the lower level dd measurements reported in Table 1a, however, more tuning for the A2 processor might diminish that further. These measurements indicate that GPFS on BGAS is well suited for applications requiring bandwidth even for transfers smaller than the GPFS block size.

When looking at IOPS on a block device, adding concurrency is very important as can be seen by a more than 2.5X increase in IOPS for small reads and writes when going from $\$MPIproc = 1$ to 4 (Figure 3; right panel). However, even these numbers do not max out the HS4 capabilities compared to when

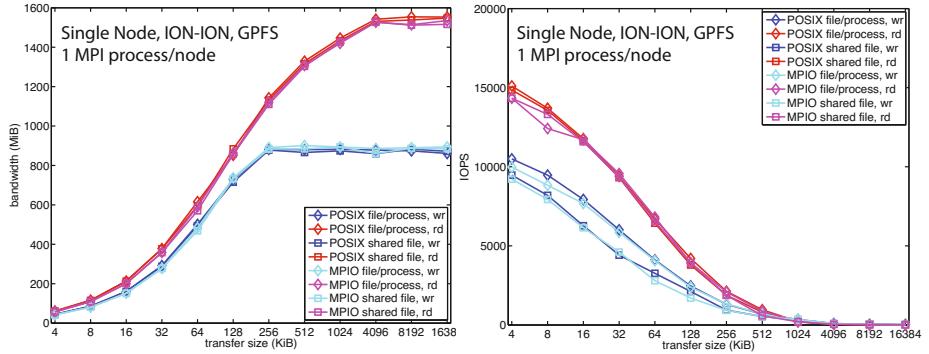


Fig. 2. Single node performance of an IOR benchmark running on BGAS ION to GPFS on flash. GPFS and IOR parameters as described in Section 5.

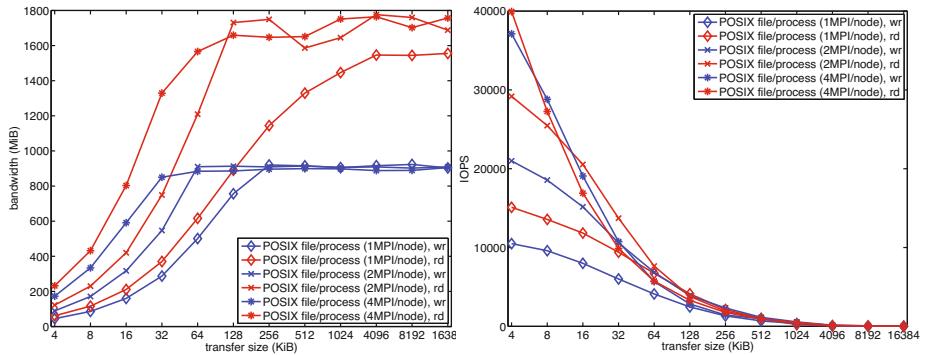


Fig. 3. Single node performance on BGAS ION as function of the number of MPI processes per node used for IOR bandwidth and IOPS measurements

using DSA directly as shown in Table 1a, which is the reason why exploring alternative access methods to block devices is desirable especially for applications bound by many small random transfers.

To demonstrate the building block approach of the BGAS technology, four different GPFS partitions were created spanning 8, 16, 32 and 64 BGAS IONs respectively. These partitions can be used stand-alone on BGAS (ION-ION) or from BGQ in optimal partitions of 512, 1024, 2048, and 4096 CNK nodes respectively (CNK-ION). Scale out performance of the BGAS technology is shown in the left panel of Figure 4. Taking the aggregate performance of the 8 node ION-ION measurement as reference, we observe essentially ideal scaling to 16, 32, and 64 IONs. This holds true for POSIX shared file access (not shown). The same absolute performance and ideal scaling is also observed when accessing BGAS from 512, 1024, 2048, and 4096 BGQ compute nodes (CNK-ION). This

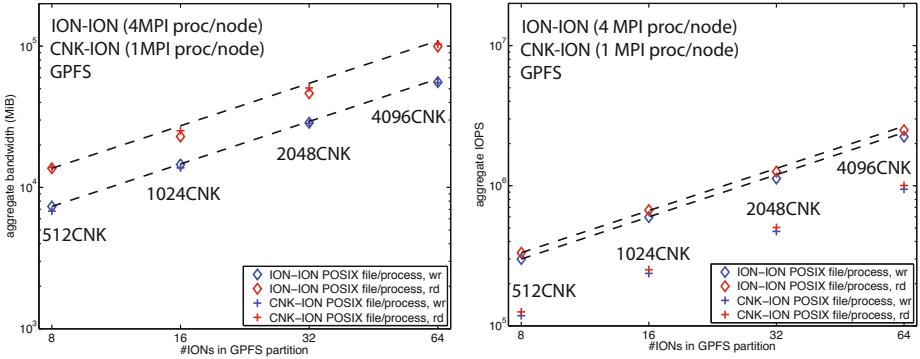


Fig. 4. Aggregate performance of BGAS as function of number of IONs partaking in the GPFS partition; Left: IOR bandwidth measurements from IONs and CNK; Right: IOR IOPS measurements from IONs and CNK

results into an aggregate bandwidth of 99,457 MiB/s read/55,748 MiB/s write for the full BGAS system.

Similarly for IOPS (see right panel of Figure 4), ideal scaling up to full system size can be observed for the stand-alone BGAS (ION-ION) scenario reaching 2,494,310 read/2,237,754 write IOPS (4 KiB) when using 64 IONs. When driving the IOPS measurement from the BGQ CNKs ideal scaling is still observed but the absolute performance only shows about 40 % of the stand-alone BGAS scenario. If one reduces the overall transferred amount of data (implying that more requests will be found in the page pool), this performance drop can be recovered fully. We therefore conclude that the GPFS implementation on BGAS is in principle capable of coping with the increased concurrency in the CNK-ION scenario but more analysis is needed to understand the ideal thread settings and pinning and optimal GPFS parameters.

Bandwidth performance was measured with the GPFS and IOR parameters described in Section 5 and `$TransferBlockSize = 4 MiB`; `$MPIproc = 4` for the ION-ION scenario and `$TransferBlockSize = 16 MiB`; `$MPIproc = 1` for the CNK-ION scenario. Analogously, aggregate IOPS were measured using `$TransferBlockSize = 4 KiB` and `$MPIproc = 4` in the ION-ION scenario and `$MPIproc = 1` in the CNK-ION scenario.

6.3 HDF5 on DSA

Measurements of bandwidth writing to a single shared file in parallel using the HDF5 driver give values of 1074, 2120, 4349, 7253 MiB/s for 1,2,4,8 nodes respectively. This shows good scaling for the node counts tested and reaches 2K IOPS for 4MB block transfers. The implementation makes use of a single DSM server per node which sends DSA put/get requests to the flash driver and MPI Send/Receive requests to the other IONs to coordinate the IO. The interface

between the two-sided MPI and one-sided DSA is the limiting factor in the performance in this implementation and can be simplified and improved by using an OFED verbs layer for both channels of communication with asynchronous transfers on both sides. Using multiple servers per node should increase the throughput by ensuring the DSA driver read/write queues are always kept full. The implementation of HDF5 directly on top of DSA represents the first step towards exposing the low level hardware access directly to applications in user-space, which in turn opens up new possibilities for the exploitation of this type of memory layer.

7 Conclusions and Outlook

We presented a particular instance of storage-class-memory, namely SLC flash, integrated at scale with an IBM BlueGene/Q supercomputer: BlueGene Active Storage. The motivation for building this system stemmed from the use-case of data-centric computing, particularly prominent in detailed brain tissue modeling and simulation as pursued by the Blue Brain Project. A prototype system has been developed and installed, and first benchmarks presented.

Particular effort has been put into exposing the properties of SCM to the application by developing a novel direct storage-class-memory access (DSA) method, based on the OFED RDMA interface. We demonstrated that it can serve as the foundation on which to build block devices to support file systems, namely GPFS, which serves as an immediate transition path for applications using legacy IO layers; we demonstrated this by achieving 99,457 MiB/s read and 55,748 MiB/s write bandwidth using 64 BGAS IONs; these results show ideal scaling to full system size. We furthermore demonstrate that the bandwidth performance is fully available to the attached BlueGene/Q compute nodes, also showing ideal scaling from 512 to 4096 nodes and identical absolute performance. We furthermore tested the systems capabilities for small (4 KiB) IOPS using GPFS, showing ideal scaling from 8 to 64 IONs and reaching 2,494,310 read and 2,237,754 write IOPS when running on 64 IONs of BGAS. Scaling holds true when driving the small IOPS from 512 to 4096 BlueGene/Q compute nodes but overall IOPS are reduced when compared to the BGAS only case.

We conclude that GPFS provides an excellent basis for bandwidth oriented access patterns and also provides a good foundation for large numbers of small IOPS with some more configuration work to do. At the same time, a user-space micro-benchmark shows that if DSA is used directly without a block device, it has the potential to boost IOPS even further. We present prototype work to expose this to applications using HDF5 and a custom driver to map it directly on top of DSA. We show first results for 8 IONs and we expect to scale this work to the entire system in the future.

Even though this prototype system is still in its early evaluation phase, the preliminary benchmark results convincingly demonstrate its potential. Further work will be undertaken to tune the software stack to get even closer to the capabilities of the hardware as indicated by the micro-benchmarks. Progressively,

more applications and workflows will be adapted to directly use the DSA and to illustrate the role of IOPS optimized storage in HPC for a real-world application use case going beyond checkpoint/restart. How to provision this versatile storage-class memory to applications will be part of the future research: one can imagine provisioning tailored storage on a per application and job basis or using BGAS's capabilities to have multiple partitions (each tailored for a particular access pattern) running concurrently.

We acknowledge that alternative solutions exist, including using DRAM-resident file systems or key-value stores (e.g. [31]), however, they cannot compete with the energy and cost efficiency of the solution presented here at the same capacity. Flash-based solutions can provide larger capacity than DRAM for the same cost while providing much more capability than a disk-based system. The relative costs of various types of memories/storage have been considered in series of papers, the most recent of which includes flash as well as DRAM and magnetic disk storage [43–45]. These papers use a criterion for deciding when there is a cost benefit to purchasing additional higher performance/higher cost storage based on frequency of access. The cross-over occurs when the cost of purchasing an additional increment of DRAM or Flash equals the cost of provisioning a disk-based system that can support the target access rate for a block of storage. Blocks whose target access rate exceed this threshold should be stored in the higher performing storage technology because the relative costs justify it. One can make an analogous argument to assess the relative energy costs of two storage technologies and some data regarding the relative power consumption of flash and magnetic disk is available [46].

Using the design point we are prototyping with the current BGAS system, integrating SCM into the Blue Gene/Q compute fabric should enable 1 PB (MLC flash) per half rack. As memory densities increase, for example with 3D stacking, increasingly large storage-class-memory fabrics with very high internal bandwidth become possible. The large internal bandwidth and the cost of moving data across a data center drives the key active storage design objective of embedding computation in the storage-class-memory array. Especially, with the ambitious roadmap toward human brain scale models, the cost efficiency of SCM has to be exploited. The current BGAS prototype, which anticipates the byte-addressability and hybrid nature of future SCM systems, offers an exciting research platform to explore data-centric computing today.

Contributions

The IBM team initiated the BGAS project, developed the respective hardware and software of the HS4 card and integration with the IONs as well as provided micro benchmarks and support. The CSCS team manages the system and led the development of the custom HDF5 driver and performed respective benchmarks. The EPFL team led the overall system integration and the present study.

Acknowledgments. We thank Carlos Aguado and Luc Corbeil for their active role of the definition and operation of the Blue Brain IV environment. We thank the members of the extended Active Storage hardware and software teams within IBM including Todd Takken, Heiko J. Schick, Ben Krill, Michael Deindl, Michael Kaufmann, Marc Dombrowa, David Satterfield, Ralph Bellofatto, Alda Ohmacht, and Uwe Fischer for their essential contributions to the Blue Gene Active Storage platform. We also thank the IBM Research CSS group for assembling the HS4 cards used for this work. Portions of the Blue Gene Active Storage development have been supported and partially funded by Argonne National Laboratory and the Lawrence Livermore National Laboratory on behalf of the U.S. Department of Energy, under Lawrence Livermore National Laboratory subcontract no. B554331. The EPFL Blue Brain Project, the Blue Brain IV system as well as parts of this study are funded by the ETH board.

References

1. Strande, S.M., Cicotti, P., Sinkovits, R.S., Young, W.S., Wagner, R., Tatineni, M., Hocks, E., Snavely, A., Norman, M.: Gordon: Design, performance, and experiences deploying and supporting a data intensive supercomputer. In: Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment: Bridging from the eXtreme to the Campus and Beyond, XSEDE 2012, New York, NY, USA, pp. 3:1–3:8. ACM (2012)
2. NNSA and US DoE - Office of Science, FastForward R&D draft statement of work (March 2013), <https://asc.llnl.gov/fastforward/>
3. Lawrence livermore, intel, cray produce big data machine to serve as catalyst for next-generation hpc clusters. Press Release (November 2013)
4. Williams, S., Waterman, A., Patterson, D.: Roofline: An insightful visual performance model for multicore architectures. Communications of the ACM 52(4), 65–76 (2009)
5. Eleftheriou, E., Haas, R., Jelitto, J., Lantz, M., Pozidis, H.: Trends in storage technologies. Bulletin of the Technical Committee on Data Engineering 33(4), 4–13 (2010)
6. Markram, H.: The blue brain project. Nature Reviews. Neuroscience 7, 153–160 (2006), PMID: 16429124
7. Hay, E., Hill, S., Schürmann, F., Markram, H., Segev, I.: Models of neocortical layer 5b pyramidal cells capturing a wide range of dendritic and perisomatic active properties. PLoS Comput. Biol. 7, e1002107 (2011)
8. Reimann, M.W., Anastassiou, C.A., Perin, R., Hill, S.L., Markram, H., Koch, C.: A biophysically detailed model of neocortical local field potentials predicts the critical role of active membrane currents. Neuron 79, 375–390 (2013)
9. Hill, S.L., Wang, Y., Riachi, I., Schürmann, F., Markram, H.: Statistical connectivity provides a sufficient foundation for specific functional connectivity in neocortical neural microcircuits. Proceedings of the National Academy of Sciences 109, E2885–E2894 (2012), PMID: 22991468
10. Herculano-Houzel, S., Mota, B., Lent, R.: Cellular scaling rules for rodent brains. Proceedings of the National Academy of Sciences of the United States of America 103, 12138–12143 (2006)

11. Kozloski, J., Sfyrakis, K., Hill, S., Schürmann, F., Peck, C., Markram, H.: Identifying, tabulating, and analyzing contacts between branched neuron morphologies. *IBM J. Res. Dev.* 52, 43–55 (2008)
12. Migliore, M., Cannia, C., Lytton, W.W., Markram, H., Hines, M.L.: Parallel network simulations with NEURON. *Journal of Computational Neuroscience* 21, 119–129 (2006)
13. Tauheed, F., Biveinis, L., Heinis, T., Schürmann, F., Markram, H., Ailamaki, A.: Accelerating range queries for brain simulations. In: 2012 IEEE 28th International Conference on Data Engineering (ICDE), pp. 941–952 (April 2012)
14. Mesnier, M.P., Wachs, M., Sambasivan, R.R., Lopez, J., Hendricks, J., Ganger, G.R.: Trace: Parallel trace replay with approximate causal events. In: Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST 2007). MCDOUGALL (2007)
15. Shan, H., Shalf, J.: Using IOR to analyze the I/O performance for HPC platforms. In: Cray User Group Conference (CUG 2007) (2007)
16. May, J.: Pianola: A script-based I/O benchmark. In: Petascale Data Storage Workshop, PDSW 2008, 3rd edn., pp. 1–6 (November 2008)
17. Frings, W., Hennecke, M.: A system level view of petascale I/O on IBM blue Gene/P. *Computer Science - Research and Development* 26, 275–283 (2011)
18. Carns, P., Harms, K., Allcock, W., Bacon, C., Lang, S., Latham, R., Ross, R.: Understanding and improving computational science storage access through continuous characterization. In: 2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST), pp. 1–14 (May 2011)
19. Xie, B., Chase, J., Dillow, D., Drokin, O., Klasky, S., Oral, S., Podhorszki, N.: Characterizing output bottlenecks in a supercomputer. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012, Los Alamitos, CA, USA, pp. 8:1–8:11. IEEE Computer Society Press (2012)
20. Lofstead, J.F., Klasky, S., Schwan, K., Podhorszki, N., Jin, C.: Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). In: Proceedings of the 6th International Workshop on Challenges of Large Applications in Distributed Environments, CLADE 2008, New York, NY, USA, pp. 15–24. ACM (2008)
21. Frings, W., Wolf, F., Petkov, V.: Scalable massively parallel I/O to task-local files. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC 2009, New York, NY, USA, pp. 17:1–17:11. ACM (2009)
22. Behzad, B., Luu, H.V.T., Huchette, J., Byna, S.: Taming parallel I/O complexity with auto-tuning. In: Gropp, W., Matsuoka, S. (eds.) SC, p. 68. ACM (2013)
23. Cohen, J., Dossa, D., Gokhale, M., Hysom, D., May, J., Pearce, R., Yoo, A.: Storage-intensive supercomputing benchmark study. Technical report, Lawrence Livermore National Laboratory (2007)
24. Park, S., Shen, K.: A performance evaluation of scientific I/O workloads on flash-based SSDs. In: IEEE International Conference on Cluster Computing and Workshops, CLUSTER 2009, pp. 1–5 (August 2009)
25. Jung, M., Kandemir, M.: Revisiting widely held SSD expectations and rethinking system-level implications. In: Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2013, New York, NY, USA, pp. 203–216. ACM (2013)

26. Zheng, D., Burns, R., Szalay, A.S.: Toward millions of file system IOPS on low-cost, commodity hardware. In: Proceedings of SC 2013: International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2013, New York, NY, USA, pp. 69:1–69:12. ACM (2013)
27. Fitch, B., Rayshubskiy, A., Ward, T., Germain, R.: Toward a general parallel operating system using active storage fabrics on Blue Gene/P. In: Computing with Massive and Persistent Data (CMPD 2008) (September 2008)
28. Fitch, B.G., Rayshubskiy, A., Pitman, M.C., Ward, T.J.C., Germain, R.S.: Using the active storage fabrics model to address petascale storage challenges. In: Proceedings of the 4th Annual Workshop on Petascale Data Storage, PDSW 2009, New York, NY, USA, pp. 47–54. ACM (2009)
29. Andersen, D.G., Franklin, J., Kaminsky, M., Phanishayee, A., Tan, L., Vasudevan, V.: Fawn: a fast array of wimpy nodes. In: SOSP 2009: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, New York, NY, USA, pp. 1–14. ACM (2009)
30. Vasudevan, V., Tan, L., Andersen, D., Kaminsky, M., Kozuch, M.A., Pillai, P.: FawnSort: Energy-efficient sorting of 10gb. Winner of 2010 10GB Joulesort Daytona and Indy categories (2010),
<http://sortbenchmark.org/fawnSort-joulesort-2012.pdf>
31. Ousterhout, J., Agrawal, P., Erickson, D., Kozyrakis, C., Leverich, J., Mazières, D., Mitra, S., Narayanan, A., Ongaro, D., Parulkar, G., Rosenblum, M., Rumble, S.M., Stratmann, E., Stutsman, R.: The case for RAMcloud. Communications of the ACM 54(7), 121–130 (2011)
32. Jung, M., Wilson III, E.H., Choi, W., Shalf, J., Aktulga, H.M., Yang, C., Saule, E., Catalyurek, U.V., Kandemir, M.: Exploring the future of out-of-core computing with compute-local non-volatile memory. In: Proceedings of SC 2013: International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2013, New York, NY, USA, pp. 75:1–75:11. ACM (2013)
33. I. B. G. team, The IBM blue gene project. IBM Journal of Research and Development 57, 0:1–0:6 (2013)
34. Chen, D., Eisley, N.A., Heidelberger, P., Senger, R.M., Sugawara, Y., Kumar, S., Salapura, V., Satterfield, D., Steinmacher-Burow, B., Parker, J.: The IBM blue Gene/Q interconnection fabric. IEEE Micro 32(1), 32–43 (2012)
35. Schmuck, F., Haskin, R.: GPFS: A shared-disk file system for large computing clusters. In: FAST 2002: Proceedings of the 1st USENIX Conference on File and Storage Technologies, Berkeley, CA, USA, p. 19. USENIX Association (2002)
36. Haring, R., Ohmacht, M., Fox, T., Gschwind, M., Satterfield, D., Sugavanam, K., Coteus, P., Heidelberger, P., Blumrich, M., Wisniewski, R., Gara, A., Chiu, G., Boyle, P., Chist, N., Kim, C.: The IBM blue Gene/Q compute chip. IEEE Micro 32, 48–60 (2012)
37. Ryu, K.D., Inglett, T.A., Bellofatto, R., Blocksome, M.A., Gooding, T., Kumar, S., Mamidala, A.R., Megerian, M.G., Miller, S., Nelson, M.T., Rosenberg, B., Smith, B., Van Oosten, J., Wang, A., Wisniewski, R.W.: IBM blue Gene/Q system software stack. IBM Journal of Research and Development 57, 5:1–5:12 (2013)
38. OFED overview. Open Fabrics Alliance Website,
<https://www.openfabrics.org/resources/ofed-for-linux-ofed-for-windows/ofed-overview.html>
39. Soumagne, J., Biddiscombe, J., Esnard, A.: Data Redistribution using One-sided Transfers to In-memory HDF5 Files. In: Cotronis, Y., Danalis, A., Nikolopoulos, D.S., Dongarra, J. (eds.) EuroMPI 2011. LNCS, vol. 6960, pp. 198–207. Springer, Heidelberg (2011)

40. Biddiscombe, J., Soumagne, J., Oger, G., Guibert, D., Piccinelli, J.-G.: Parallel Computational Steering for HPC Applications using HDF5 Files in Distributed Shared Memory. *IEEE Transactions on Visualization and Computer Graphics* 18, 852–864 (2012)
41. Ior: Github repository, <https://github.com/chaos/ior>
42. Mpich2: Official website, <http://www.mcs.anl.gov/research/projects/mpich2staging/goodell/>
43. Gray, J., Putzolu, F.: The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for cpu time. *SIGMOD Rec.* 16(3), 395–398 (1987)
44. Gray, J., Graefe, G.: The five-minute rule ten years later, and other computer storage rules of thumb. *SIGMOD Rec.* 26(4), 63–68 (1997)
45. Graefe, G.: The five-minute rule 20 years later: and how flash memory changes the rules. *Queue* 6, 40–52 (2008)
46. Gray, J., Fitzgerald, B.: Flash disk opportunity for server applications. *Queue* 6, 18–23 (2008)

Orthrus: A Framework for Implementing Efficient Collective I/O in Multi-core Clusters

Xuechen Zhang¹, Jianqiang Ou², Kei Davis³, and Song Jiang²

¹ Georgia Institute of Technology, Atlanta, GA, USA

² Wayne State University, Detroit, MI, USA

³ Los Alamos National Laboratory, Los Alamos, NM, USA

Abstract. Optimization of access patterns using collective I/O imposes the overhead of exchanging data between processes. In a multi-core-based cluster the costs of inter-node and intra-node data communication are vastly different, and heterogeneity in the efficiency of data exchange poses both a challenge and an opportunity for implementing efficient collective I/O. The opportunity is to effectively exploit fast intra-node communication. We propose to improve communication locality for greater data exchange efficiency. However, such an effort is at odds with improving access locality for I/O efficiency, which can also be critical to collective-I/O performance. To address this issue we propose a framework, *Orthrus*, that can accommodate multiple collective-I/O implementations, each optimized for some performance aspects, and dynamically select the best performing one accordingly to current workload and system patterns. We have implemented Orthrus in the ROMIO library. Our experimental results with representative MPI-IO benchmarks on both a small dedicated cluster and a large production HPC system show that Orthrus can significantly improve collective I/O performance under various workloads and system scenarios.

1 Introduction

Petascale HPC systems are current, and issues critical to the delivery of exascale systems in the next decade are being actively identified and studied [1]. A fundamental approach to reaching very high compute capacity is to employ increasingly large numbers of compute nodes, each with increasingly more CPU cores. A major challenge on such a technical path is application scalability. For parallel programs or system facilities that need global communication and coordination, the benefit of increased system scale cannot be fully realized if locality in the operations is not carefully exploited. In this paper we take collective I/O, a widely-used and performance-critical facility in the MPI I/O library, as a representative case for studying how to introduce communication locality into its implementation, and to demonstrate a strategy to effectively exploit locality.

1.1 Potential Challenges in the Performance of Collective I/O

Collective I/O is a technique commonly employed in MPI programs to coordinate and reorganize requests from the multiple processes of a program before sending them to the data nodes. The idea is simple: if each process needs to access one or a number of segments of data in a file domain, and collectively these processes access all or a major part of the domain, it is more efficient to have each of (a subset of) the processes access a large and mostly contiguous segment of data in the domain. In this way there can be fewer requests to the data nodes for higher data access efficiency.

There are two factors that may compromise performance. One is the potentially high cost of the data exchange that is central to collective I/O. In collective I/O a process, the aggregator, is responsible for access to non-overlapping sections of a file, which is called the aggregator's file realm. An aggregator's file realm may include data requested by other processes, so a data exchange phase is required in addition to the I/O phase. For reads the data exchange happens after the aggregators have retrieved the data into their respective buffers. Data exchange for writes, wherein the aggregator collects data for writing, precedes the I/O phase. The overhead of data exchange can be significant if a large amount of data needs to be exchanged, especially when these aggregators and the processes communicating with them are on different compute nodes. If there are multiple cores on each node—the common case in today's HPC systems—the ideal scenario in terms of minimizing the cost of data exchange is to have each aggregator only be responsible for accessing data for processes on the same node and so have all data exchange occur within individual nodes. However, this assignment of file realms to aggregators may compromise the efficiency of I/O operations on the data nodes—the second factor that affects collective-I/O performance.

When data is stored on hard disks—currently the dominant storage devices in HPC systems—disk efficiency determines I/O efficiency, especially for requests that are not very large. For a disk to efficiently serve a set of temporally proximal requests, the order in which they reach the disk, which largely determines the order in which they are served, is a major factor in the disk's efficiency. Because a hard disk relies on disk-head seek and disk-platter rotation to reach requested data, arrival of requests in ascending disk order minimizes its mechanical operations and maximizes its throughput. However, if the requests to a disk are issued from different aggregators, there is no way to ensure ascending order unless the involved aggregators synchronize their issuance, which can be excessively expensive, especially in a large-scale system. The ideal scenario in terms of maximizing disk efficiency is to have all requests to a disk (or data node) be issued by one aggregator that can send them in the ascending order according to requested data's offsets in the file [35].

1.2 Data Exchange Efficiency vs. Disk Service Efficiency

As described, a strategy for assigning file realms to aggregators to achieve high efficiency for data exchange is at odds with one for achieving high disk service

efficiency. For example, data on a disk can be requested by processes on different compute nodes, and if an aggregator's file realm is only the data requested by processes on the same node to avoid inter-node data exchange, the hard disk would receive requests from different aggregators and so risk loss of efficiency.

It is not clear how to define a file realm assignment that is optimized for both disk access and data exchange, and an implementation of one strategy or the other is not expected to perform well for collective I/O with diverse patterns. However, we can *dynamically* determine how biasing the trade-off will affect collective-I/O service time and accordingly apply the appropriate strategy to improve efficiency.

We propose a general framework that allows multiple collective-I/O implementations, each optimized for one or more performance aspects, e.g. I/O pattern or relative hardware performance characteristics, to co-exist in a library. Based on a prediction of which would perform best for the current I/O pattern and system load, and applying it, the framework essentially provides MPI programmers a collective-I/O library that adapts to access pattern changes and other dynamic system characteristics. This framework, named *Orthrus*, extensibly accommodates multiple collective-I/O implementations.

1.3 The Challenge and Our Contributions

A major challenge for the framework to achieve effectiveness is in the dynamic prediction of the performance of the various candidate implementations in a given scenario. Both off-line modeling and on-line simulation are unlikely to provide predictions accurate enough to distinguish the candidates. One reason is that both modeling and simulation need not only the workload information (such as number of aggregators, request count, volume and distribution of requested data across data nodes) but also the system information, especially that about data nodes, such as number of data nodes, type of storage devices, and actual data layout on the devices. Furthermore, the information may change from one run to another. In general there will be system dynamics that are impossible to predict such as communication traffic generated by other programs on the compute-node side, or streams of I/O requests from unrelated programs sharing the same data nodes. Some information, such as data layout, is simply not available to user-level programs, including the MPI libraries. Much of the information is hard to accurately capture in an on-line manner, let alone for the modeling or simulation facilities to use for accurate and on-the-fly prediction.

We propose a simple, efficient, accurate, and portable method for selecting the best performer for a collective-I/O operation. To this end it does not take any workload or system information as input and does not involve any complex modeling or simulation for performance prediction. The key technique is to use performance examination, rather than performance modeling, in the prediction. Each candidate collective-I/O implementation is given opportunities to demonstrate its performance, which is recorded for comparison and selection. In this way all of the hard-to-capture information is distilled in the actual performance of a candidate's examination run, which provides an accurate prediction of the

performance that would be exhibited should the candidate be selected for executing the collective I/O in the near future. In summary we make the following contributions.

- We propose a framework for accommodating multiple collective I/O implementations, each optimized for some class of I/O patterns, that dynamically selects from them for best performance, and is thus adaptive to changing workloads and system dynamics.
- We develop an efficient mechanism to evaluate the performance of collective-I/O implementations so that the best performer can be identified and deployed on the fly. With this mechanism parallel I/O developers need not commit to compromising trade-offs in a collective-I/O implementation. Instead, they can implement schemes optimized for specific workloads or system characteristics and simply plug them into the framework, where their performance advantages would be realized whenever their targeted characteristics appear.
- We experimentally evaluate our implementation of the *Orthrus* framework in the ROMIO library, and provide a detailed analysis of the results. Our experimental results with representative MPI-IO benchmarks on both a small dedicated cluster and a large production HPC system show that *Orthrus* can significantly improve the I/O throughput of storage systems.

2 The Design of Orthrus

The objective of *Orthrus* is to provide a framework for implementing a high-performance collective I/O library that can adaptively achieve high I/O efficiency with various workload patterns and system setups. It represents a deviation from the traditional practice of attempting to build a single, highly-versatile implementation to efficiently handle different kinds of workloads. Realizing that a monolithic design tends to lack flexibility through extensibility, we subdivide the effort. A key element is the design of a framework ready for multiple implementations to plug into. The other elements are the provision of collective-I/O implementations. One advantage of this approach is making the second effort open to other practitioners in the HPC community. For workloads that are of particular importance to them but not handled well by existing implementations, they can develop new implementations specific to their workloads (and machine architectures) and plug them into the framework without the concerns associated with the traditional approach, such as compromising the existing implementation or complicating the design. In this section, we describe our efforts on these two fronts.

2.1 The Orthrus Framework

The crux of the *Orthrus* strategy is knowing how each candidate implementation would likely perform if it were used to execute a collective-I/O function call.

Flash-io	GTS
<pre> do i = 1, nvar record_label = unklabels(i) unk_buf(1,1:nxb,1:nyb,1:nzb,:) = unk(i,nguard+1:nguard+nxb, + nguard*k2d+1:nguard*k2d +nyb, + nguard*k3d+1:nguard*k3d +nzb, :) ... call h5_write_unknowns(file_id, + + 1, + 1, + nxb, + nyb, + nzb, + 0, + maxblocks, + unk_buf, + record_label, + lnblocks, + tot_blocks, + global_offset) ... enddo </pre>	<pre> do istep=istart,mstep do irk=1,2 ... ! push ion ! CALL PUSHI CALL PUSHION ... ! I/O operations CALL SNAPSHOT ... enddo ! MAIN TIME LOOP </pre>

Fig. 1. *Flash-io* periodically writes out checkpointing data using the function *h5_write_unknowns()*. In the *GTS* code, both checkpoint and visualization data are written back to the disks by the *SNAPSHOT* function. Thousands of iterations are typically taken in their execution in production HPC runs. Different I/O transports can be specified, e.g. MPI collective I/O, in the code when the ADIOS [25] library is used.

As mentioned, *Orthrus* examines the performance of each candidate. Certainly in the library one application-level call cannot be executed more than once: the overhead would be excessive and the caching effect would invalidate the performance results of all but the first call. However, with different candidates executing different calls we would need to ensure that their performance results were comparable by invoking these candidates within the same workload, i.e., have the data requested by the calls have the same pattern. In addition, the performance examination period must constitute only a small fraction of the total collective-I/O time because many candidates might not provide efficient I/O service.

Our solution is based on the assumption that a collective-I/O function call is usually in a loop of many iterations. For our purpose this is a reasonable assumption, otherwise the call would be executed only one or a few times, either generating requests for a small aggregate amount of data, whose I/O performance would likely be insignificant for the entire program's performance, or generating a few large requests, each for a large amount of data, whose I/O performance is generally not at issue. Through experimental observations, we also find that many scientific applications, such as *Flash-io* [15] and *GTS* [28], use loop statements to carry out I/O operations for checkpointing or dumping visualization data during their entire execution periods. Their I/O patterns are outlined in Figure 1.

We also assume a consistent access pattern across the loop iterations in the execution of a collective I/O function call. The iteration's data access can be

characterized by a number of factors, including name of accessed file, number of processes issuing I/O requests, number of requests, request sizes, and their offsets in the file. While offsets of requests issued in different iterations cannot be compared directly, for characterization we define a so-called relative offset and use this as a signature to identify similar spatial access patterns quickly. Suppose that in the execution of iteration k of a loop, a collective I/O call produces n I/O requests that sorted according to the offsets of their requested data are $[R_1, R_2, \dots, R_n]$. If the size and file offset of request R_i are $Size_i$ and $Offset_i$, respectively, the relative offset of these requests is $rel_offset = \frac{\sum_{i=2}^n [Offset_i - (Offset_{i-1} + Size_{i-1})]}{n-1}$. We refer to these values collectively as the call's *signature*. Two collective I/O calls are deemed to have the same pattern and comparable performance results only when their signatures are the same. I/O signature-based pattern identification has been effectively used in previous works, for example in data prefetching [4].

The assumption of consistency provides us with two capabilities. One is that we can use the observed performance of one execution of a collective I/O call as an estimate of its performance in subsequent executions. Second, it allows us to identify calls from the same program statement—we do not assume the availability of the program's source code or the ability to instrument it to explicitly notify the ROMIO library in which *Orthrus* is implemented.

Under these assumptions *Orthrus* carries out its operations as follows. When a collective-I/O call is made, first its signature is compared to the signature of the previous call. If they are not the same, a new candidate examination period is started. If they are the same, then either the system is in a candidate examination period, in which case *Orthrus* keeps testing a candidate implementation, or the system uses the currently selected candidate implementation to execute the call. When a new candidate examination period is started, calls are serviced by the candidates in rotation, each for a fixed number of times (three by default), as long as the signature does not change. The throughput of each candidate is computed as an average to minimize the effects of transient changes in the execution environment. When the examination period ends without a change in signature, the candidate with the highest throughput is selected to execute the calls until the signature changes.

During a regular execution period, even if the collective-I/O calls do not change their access pattern, the dynamic system environment, such as communication and I/O request traffic initiated by other programs, could change, and accordingly the best performing candidate might change. To detect such system variations we monitor throughput of the selected candidate. If its deviation exceeds a certain threshold (15% by default) of the average recorded in the examination period the system returns to the examination mode.

2.2 Candidate Collective-I/O Implementations

To test our idea we implemented two simple, contrasting strategies. One, *core-first*, constrains data exchange to be intra-node. In the current implementation of *core-first* there is one aggregator per compute node to represent all the processes

running on that node. We choose as the aggregator the process in that node requesting the largest amount of data. The processes' I/O requests are collected by the aggregator, which then sorts them in the ascending order according to the offsets of the requested data. When the number of cores in each node is relatively small (e.g. less than 100) using one aggregator is sufficient to handle the data management. With more cores multiple aggregators might be needed and a more sophisticated algorithm could be developed to distribute the load.

The second strategy, *disk-first*, is designed to maximize disk efficiency. It sets up the same number of aggregators as the number of data nodes, each collecting and sending requests to one data node, sorting the requests as *core-first* does. However, *disk-first* ensures that each data node receives requests in well-ordered sequences, while in *core-first* each data node receives requests from multiple (possibly all) aggregators, and the order of requests from different aggregators is essentially random. While maintaining an equal number of aggregators as data nodes allows the data nodes to receive fully sorted sequences of requests, it may limit the I/O bandwidth if the number of data nodes is small. One solution is having multiple aggregators coordinated to access a data node [35].

Though the existing collective-I/O implementation in the ROMIO library does not attempt to reduce communication or maximize disk efficiency, it does reduce the number of requests. One method used is to designate a contiguous file domain to each aggregator. As an I/O request to operating system must be for a contiguous segment of data, this helps reduce the number of requests. However, in ROMIO such a contiguous file domain may contain holes, or gaps in the domain that are not requested. Such holes break the contiguity, and also breaks a potentially large request into multiple smaller ones. To remedy this ROMIO uses data sieving [22] to remove the holes to yield one large request. The benefits of having a smaller number of large requests include reduced request processing overhead and increased disk efficiency. As the *core-first* and the *disk-first* schemes do not use *list I/O* [10] to pack requests, the number of requests to the kernels of the data nodes is not reduced. When the requests are very small (even if they are well ordered), ROMIO's increased disk efficiency through data sieving can be still substantial in comparison. Therefore, as a third strategy we plug the ROMIO implementation into the *Orthrus* framework. For fair comparison we set up only one aggregator per compute node in this ROMIO instantiation.

We emphasize that we are *not* attempting to provide a comprehensive or near-optimal set of strategies in this paper. Rather, the purpose of our choices is to provide a simple set of contrasting strategies to demonstrate the effective adaptivity of the *Orthrus* framework. We include the ROMIO implementation as a candidate strategy as a touchstone of realism—if the performance of these three strategies can exceed that of ROMIO alone, our case is much stronger than it might be if only synthetic strategies were used.

3 Performance Evaluation

The *Orthrus* framework was implemented as an extension of ROMIO in mpich2-1.4.1, a widely used MPI-IO implementation. It currently hosts three

collective-I/O implementations: *core-first*, *disk-first*, and ROMIO’s implementation (*ROMIO* hereafter). In this section we evaluate the framework with these three self-contained and independent collective-I/O implementations. We specifically answer three questions: (1) When does *Orthrus* outperform the existing ROMIO implementation? (2) How effective is *Orthrus* in identifying the best performing candidate implementation? (3) Does *Orthrus* work as expected when the execution environment changes dynamically?

3.1 Experimental Setup

To evaluate the performance of *Orthrus* we used a dedicated cluster allowing full control of the running environment. This allowed collection of low-level performance statistics and controlled injection of interference. The cluster consisted of 11 compute nodes and six data nodes. Each compute node was equipped with an 8-core L5410 2.33 GHz Intel Xeon CPU and 64 GB DRAM. Each data node had a 2.13GHz Intel Core 2 CPU, 2 GB DRAM, and one 500 GB SATA hard disk (WDC WD5000AAKS). We used PVFS2 version pvfs-2.8.2 for parallel storage management with its default 64KB striping configuration [19]. All of the nodes were connected through a 1 Gbps Ethernet network. We used MPICH2-1.4.1 [27] compiled with ROMIO to generate MPI executables. The processes were evenly distributed across the nodes and their respective cores. Each node ran a CentOS Linux distribution with kernel-3.3.6. The CFQ I/O scheduler [5] was used for the disks as is standard practice.

We used four benchmarks in the evaluation: *Matrix*, *Noncontig*, *Hpio*, and *Flash-io*. The first one simulates a common pattern of matrix access in scientific applications. The next two have access patterns that may pressure both the disks and the communication, having non-contiguous I/O patterns and the need to access a substantial amount of data that may be involved in the data exchange (usually more than 10 MB in one collective operation). *Flash-io* periodically writes checkpoint and visualization data using HDF5 [17] with parallel I/O. Following we describe experiments with each benchmark.

3.2 Matrix

In the *Matrix* benchmark data are viewed as elements of a two-dimensional matrix with columns evenly distributed among processes. The matrix is serialized by row-major order. Each process accesses a set of contiguous columns, and the data in one row of these columns constitutes a block as illustrated in Figure 2. In each collective-I/O operation all processes access the same number of contiguous blocks, the *access depth*, in their respective columns. If ROMIO is used, the total amount of data accessed in a single collective-IO operation is the product of the block size, access depth, and number of processes. Because any two blocks accessed in an operation by a process are not adjacent, they are in two separate requests if *core-first* or *disk-first* is applied.

We first run the benchmark with 64 processes and 4-block access depth and vary the block size between 4 KB to 4096 KB. Figures 3(a) and 3(b) show the I/O

	P_0	P_1	P_2
Access Depth	[0 1 0] [0 0 1] [0 0 0]	[1 0 0] [0 0 0] [0 0 0]	[0 0 0] [0 0 0] [0 0 0]
	[0 0 0] [0 0 0] [0 0 0]	[0 1 0] [0 0 1] [0 0 0]	[1 0 1] [0 0 0] [0 0 0]
	[0 0 0] [0 0 0] [0 0 0]	[0 0 0] [0 0 0] [0 0 0]	[0 1 0] [0 0 0] [0 0 0]
	[0 0 0] [0 0 0] [0 0 0]	[0 0 0] [0 0 0] [0 0 0]	[0 0 0] [0 0 0] [0 0 0]

Fig. 2. The access pattern of the *Matrix* benchmark. Data in a dotted rectangle comprises a block, and the set of blocks in a solid rectangle are accessed by a process (P_0 , P_1 , or P_2) in each collective operation when the access depth is 3. The three contiguous rows in a shaded area are the total amount of data accessed by one collective I/O operation.

throughputs of the benchmark for read and write access, respectively. *Orthrus* increases the throughputs on average by 21% and 36% for reads and writes respectively, compared to the stock ROMIO library. This increase is greater up to a block size of 64 KB because the file striping unit size is 64 KB and blocks (requests) no greater than 64 KB give *disk-first* more opportunity to improve disk efficiency. Once blocks are larger than 64 KB the request size usually stays at 64 KB and improvement does not increase. Interestingly, the throughputs of both ROMIO and *Orthrus* are reduced for block sizes of 256 KB and 1024 KB. This is caused by unsynchronized service of a large request that is spread over multiple data nodes. Figure 3 also shows the throughputs with *disk-first* and *core-first*. When the block size is small *disk-first* performs better than *core-first* because it improves disk access locality. However, with increasing block (request) size this advantage is less significant because the inter-node communication cost increases with the increase of the total amount of accessed data. Thus the performance gap between *disk-first* and *core-first* is reduced with increase of block size either before or after it reaches 64 KB. Finally, with 4096 KB blocks *core-first* performs better than *disk-first*, and *Orthrus* chooses *disk-first* over *core-first*.

In the second experiment we vary the access depth from 4 blocks to 128 blocks with a fixed block size of 16 KB and process count of 64. Figure 4 shows the results of *Orthrus* selecting different schemes for executing collective I/O. When the access depth is four blocks the amount of data accessed in one collective I/O operation is only 4 MB. In such a scenario the order of requests issued to the data nodes determines the I/O efficiency, making *disk-first* outperform ROMIO by 20%. However, when the access depth is 128 blocks, the data communication volume is increased to 128 MB and the cost of data exchange increases accordingly. For example, for one ROMIO collective I/O with an access depth of 4 blocks the data exchange time accounts for 28% of its service time. When data exchange accounts for 41% with an access depth of 128 blocks, *Orthrus* selects

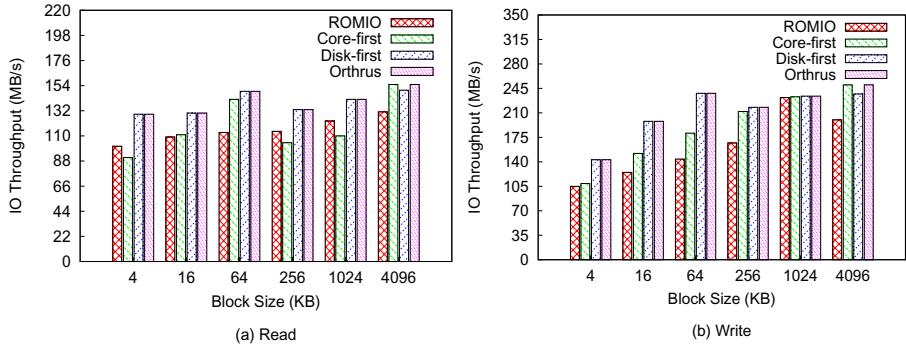


Fig. 3. Throughput of *Matrix* as a function of block size, which is also request size for *disk-first* and *core-first*, increasing from 16 KB to 4096 KB

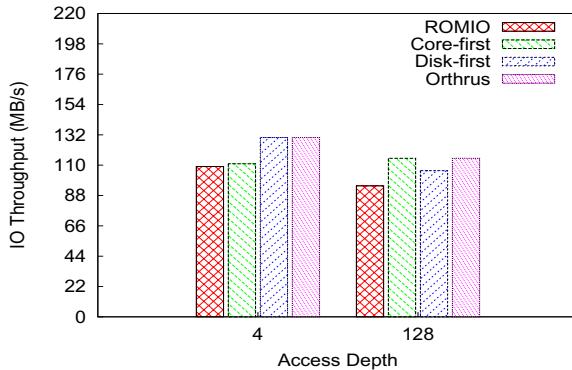


Fig. 4. Throughputs of the *Matrix* benchmark with different access depths

core-first and reduces this to 17%. Using *core-first* compromises I/O efficiency, which is why its throughput is not significantly greater than that of *disk-first*.

3.3 Noncontig

Noncontig was developed at Argonne National Laboratory [29,23]. In the experiments we configure the benchmark to simulate noncontiguous data access using the *vector* MPI derived data type. *Noncontig* uses *elmtcount* to describe size of a contiguously accessed data chunk. Each element is an MPI_LINT, which is four bytes on our system, so the request size without using ROMIO collective I/O is $4 * elmtcount$. We adjusted *veclen*, a parameter which describes the number of data chunks in a vector, so that the total accessed data size is less than 20 GB. We ran *Noncontig* with 64 processes and *elmtcount* ranging from 16 to 16384.

Table 1. Hard disk throughputs with random read and write requests of various sizes

	2 KB	4 KB	8 KB
Random Read (MB/s)	1.0	4.1	10.2
Random Write (MB/s)	0.7	0.8	2.0

Figures 5(a) and 5(b) show read and write throughputs, respectively, with increasing *elmtcount* for each of the schemes when the data are accessed from the hard disks. *Orthrus* consistently tracks the highest achievable throughputs produced by the three candidate implementations. When request sizes are 16, 64, and 256 times *elmtcount*, i.e. the effective request sizes are 64, 256 and 1024 bytes, which are smaller than the memory page size (4 KB), issuing such small random requests to disks significantly compromises disk efficiency. This explains the low throughput with *core-first* which does not form large requests or ensure that data nodes receive sorted request streams. Interestingly, when the request size is small, i.e. *elmtcount* is 16/64/256 bytes for reads and 16/64/256/1024 bytes for writes, the throughput of ROMIO is greater than that of *disk-first* which is specifically optimized to produce fully sorted requests at each data server. ROMIO achieves the performance advantage by transforming a large number of small requests into one large one, which can be more effective when the request count is very large and requests are very small. As *elmtcount* increases to 1024 for read (4096 for write), the amount of accessed data in one collective-I/O operation is very large and the data exchange cost becomes significant. For read at 1024**elmtcount*, 67% of the operation's service time is spent on data exchange. By using *core-first* this is reduced to 6% and we observe that the throughput of *core-first* starts to exceed that of ROMIO. When *elmtcount* reaches 16,384 the data exchange time is 90% of the service time while for *core-first* it is only 2%. *Core-first* has a clear performance advantage over both ROMIO and *disk-first*, and *Orthrus* is accordingly represented by *core-first*. Figure 5 also shows that in general read has higher throughput than write. This is because hard-disk write bandwidth is lower than read bandwidth when the request size is not large (Table 1), and the fact that the benchmark does I/O synchronization after every I/O operation, which effectively disables the optimization of system write-back.

3.4 Flash-io

We used macro-benchmark *Flash-io* to further evaluate *Orthrus*. *Flash-io* is designed to provide a controlled environment for tuning the I/O performance of the multi-scale multi-physics simulation code FLASH [20], so the benchmark has an identical data access pattern to the original code and the performance improvements made to the benchmark can be realized by the FLASH application. In the benchmark three data files are generated using HDF5 and MPI-IO libraries—a checkpointing file (*chk*), a plot file with centered data (*plt_cnt*), and a plot file with corner data (*plt_crn*). Table 2 gives the times for the collective I/O operations for each of the files. The I/O characteristics of the benchmark

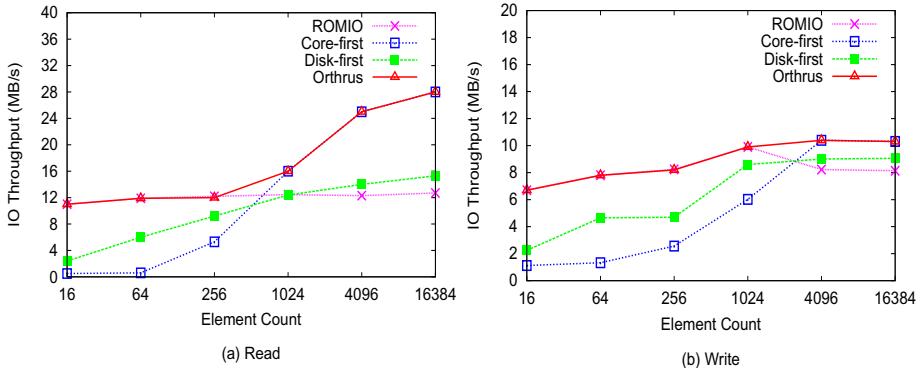


Fig. 5. Throughput of the *Noncontig* benchmark with increasing *elmtcount* when data is accessed from the hard disks

can be changed by setting the maximum number *MAXBLOCK* of blocks per processor, the number *IONMAX* of fluids to track, and the size *nxb/nyb/nzb* of the grid blocks. Two configurations are used in experiments. Configuration A has *MAXBLOCK*=1500, *IONMAX*=100, and *nxb/nyb/nzb*=1; Configuration B has *MAXBLOCK*=600, *IONMAX*=100, and *nxb/nyb/nzb*=8. We ran the benchmark with 64 processes. As shown in Table 2, the I/O times of *chk* are reduced in both Configurations A and B by 47.2% and 26.4%, respectively. Although the performance improvements are significant in both scenarios, the explanations are different. For Configuration A the average request size is 10 KB because of the small size of grid blocks. As a result, requests issued by each of the collective I/O aggregators are not large enough, and I/O performance hinges on the service order of the requests on the disks. By choosing *disk-first*, *Orthrus* allows each data node to receive the requests in ascending order, ameliorating the performance bottleneck of disk efficiency. Unlike Configuration A, processes in Configuration B issue much larger requests (2 MB), making the communication cost a dominant performance issue. By choosing *core-first*, *Orthrus* can effectively reduce the amount of inter-node communication. Another observation is that *Orthrus* incurs about 5.2% and 9.8% overhead for accessing *plt_cnt* and *plt_crn*, respectively, in Configuration A. We determined that this is because there are only four loop iterations when writing the plot files, more than nullifying the performance advantage of *Orthrus* with the overhead of examining collective I/O candidates.

3.5 Orthrus in a Dynamic Execution Environment

To analyze how *Orthrus* responds to changes in the run-time environment during a program's execution we ran *Matrix* with 32 processes and 64 KB block size to read a 10 GB file and injected a high volume of inter-node communication during its execution. Specifically, from the 60th second of *Matrix*'s execution we ran the *FT* program (discrete 3D fast Fourier Transform) from the NAS

Table 2. Collective I/O times for two different configurations of *Flash-io*. Configuration A: MAXBLOCK=1500, IONMAX=100, and nxb=1; Configuration B: MAXBLOCK=600, IONMAX=100, nxb=8. *chk* represents I/O times for checkpointing, *plt_cnt* and *plt_crn* are times for writing centered and corner plot file, respectively.

Exp	Policy	chk(s)	plt_cnt(s)	plt_crn(s)
Configuration A	ROMIO	22.8	0.38	0.51
	Orthrus	12.03	0.4	0.56
Configuration B	ROMIO	174	4.2	5.9
	Orthrus	128	3.3	3.7

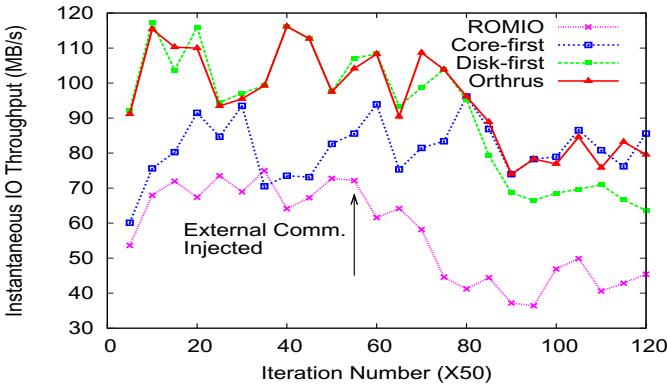


Fig. 6. Instantaneous throughput measured for the schemes after execution of every 250 iterations. The FT program starts executing at the 60th second.

Parallel Benchmarks (NPB) [30,16] with 32 processes to generate inter-node all-to-all communication. Figure 6 shows the instantaneous I/O throughput of *Matrix*, at each multiple of 250 iterations, using each of the four strategies. Initially *Orthrus* selects *disk-first* as it did in the experiment in Section 3.2. With the injection of the external communication traffic the inter-node bandwidth available to *Matrix* is reduced and its data exchange becomes more expensive. Accordingly the throughputs of ROMIO and *disk-first* are reduced by up to 46% and 60%, respectively, and *core-first* shows its advantage. As shown in Table 3, *core-first* generates a much smaller number of IP segments as reported

Table 3. Average number of incoming and outgoing IP segments transmitted at each compute node during the execution of *Matrix* with different collective-I/O schemes. Statistics are collected by reading networking stats from /proc/net/snmp.

Schemes	ROMIO	<i>core-first</i>	<i>disk-first</i>
# of In Segs.	701,600	337,257	701,056
# of Out Segs.	1,385,008	322,884	1,412,066

by `/proc/net/snmp`. When *Orthrus* detects the throughput degradation with *disk-first* it re-evaluates the candidates and responsively switches to *core-first*.

4 Related Work

Orthrus considers both the efficiency of storage devices and the heterogeneity of process communication, whereas most of other extant works consider only one of them. We classify these works into two categories, one seeking to optimize I/O streams for greater spatial locality, the other to reduce inter-node communication but preserve intra-node communication in multi-core clusters.

Optimizing Request Pattern to Improve Spatial Locality. Storage systems, especially those using hard disks for direct data access, usually cannot sustain high I/O throughput with small requests for random data from a large number of parallel processes. Many techniques have been proposed to improve requests' spatial locality by transforming them into large sequential requests, such as *data sieving* [11] in ROMIO. There are variants of collective I/O recently proposed to take account of on-disk data layout, including Resonant I/O [35] and reducing locking overhead [24] during request transformation. As *Orthrus* is a framework for dynamically selecting the best performing implementation, these variants are ready to be ported into *Orthrus* which would then inherit their performance attributes. In fact, *disk-first* is a simplified version of Resonant I/O.

To provision a framework with adaptivity to accommodate workload and system dynamics to maximize performance an effort has been made using data sieving [8]. Instead of conducting actual test runs to select the best performer among multiple candidates they a model is used to predict the performance of candidates in various settings. The model takes system/workload specifics such as disk seek time, system call time, request size, and network bandwidth as inputs. In this endeavor not only is constructing an accurate model itself a significant challenge, but also the required inputs can change and may not be (immediately) available.

Parallel I/O Optimization for Multi-core CPUs. Adoption of multi-core processors in clusters introduces non-uniform communication costs [2,13]. In such a cluster the cost of inter-node communication can be an order of magnitude higher than for intra-node communication. Furthermore, communication between cores on the same chip can be much faster than communication between chips on the same socket. Zhang et al. proposed to use the ratio of inter-node and intra-node communication costs to represent the performance effects of process affinity, which determines how processes are mapped to cores on the same or different nodes [32]. Because the implementation of ROMIO collective I/O incurs expensive all-to-all communication in the data exchange phase, Cha et al. studied the effect of aggregator assignment in the collective-I/O implementation and proposed to optimize the placement of aggregators to reduce both inter-node and intra-node communication costs [12]. Chaarawi et al. proposed a

scheme for selecting the number of aggregators based on consideration of process topology, file view, and the actual amount of data requested [14]. A study by Zou et al. found that parallel I/O performance can be compromised by lack of affinity between the core receiving an I/O interrupt and the process serving the interrupts [34]. Their proposed interrupt-scheduling scheme recovers the loss of data locality on private caches of multi-core CPUs. In the design of *Orthrus* we also consider affinity by having the option of aggregators only serving data requested by processes on the same nodes.

Self-adapting Techniques. *Orthrus* essentially adopts a self-adapting technique for improving I/O performance. Similar techniques have been used in other domains. For example, the load balancer of CHARM++ takes both computation and communication patterns into account at run-time to inform object migration for improving application scalability [6,33]. To match DRAM page size and CPU cache size, the CMSSL library contains routines for automatic selection of optimal parameters, such as loop order and operator alignments, for matrix multiplication in both local and global scopes [18]. Benkert et al. uses an empirical approach for MPI communication auto-tuning with ADCL library [3]. In comparison, *Orthrus* introduces a self-adapting technique to improve collective I/O performance and demonstrates its feasibility and effectiveness, which may inspire more innovative applications of the technique to address I/O issues in large-scale HPC systems.

5 Conclusion

We have presented the design and implementation of *Orthrus*, a framework for hosting multiple collective-I/O implementations and adaptively selecting the one that provides the highest I/O throughput according to current workload and system dynamics. Instead of attempting to optimize an existing collective-I/O implementations, or develop a new one from scratch, we demonstrate an open framework allowing multiple implementations to compete with, and complement, each other. This represents a unique approach that we have shown to be effective in the context of a multi-core cluster that has both I/O and communication efficiencies to optimize. More abstractly, it suggests a general approach for dynamic selection of multiple performance-optimization strategies where accurate modeling or simulation would be infeasible or non-portable. Experiments with the prototyped *Orthrus* in the ROMIO library show that it can improve the throughput of collective I/O under various workloads and system scenarios by up to several times.

Acknowledgments. This work was supported by US National Science Foundation under CAREER CCF 0845711, CNS 1117772, and CNS 1217948. This work was also funded in part by the Accelerated Strategic Computing program of the US Department of Energy. Los Alamos National Laboratory is operated by Los Alamos National Security LLC for the US Department of Energy under contract DE-AC52-06NA25396.

References

1. The Opportunities and Challenges of Exascale Computing,
http://science.energy.gov/media/ascr/ascac/pdf/reports/exascale_subcommittee_report.pdf/
2. Alam, S., Barrett, R., Kuehn, J., Roth, P., Vetter, J.: Characterization of Scientific Workloads on Systems with Multi-core processors. In: IEEE International Symposium on Workload Characterization (2006)
3. Benkert, K., Gabriel, E.: Empirical Optimization of Collective Communications with ADCL. In: High Performance Computing on Vector Systems 2010 (2010)
4. Byna, S., Chen, Y., Sun, X., Thakur, R., Gropp, W.: Parallel I/O Prefetching Using MPI File Caching and I/O Signatures. In: ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (2008)
5. Carroll, A.: Linux Block I/O Scheduling (2007),
<http://www.cse.unsw.edu.au/aaronc/iosched/doc/sched.pdf>
6. Parallel Languages/Paradigms: Charm ++ - Parallel Objects,
<http://charm.cs.uiuc.edu/research/charm/>.
7. Ching, A., Choudhary, A., Liao, W., Ward, L., Pundit, N.: Evaluating I/O Characteristics and Methods for Storing Structured Scientific Data. In: IEEE International Parallel & Distributed Processing Symposium (1996)
8. Lu, Y., Chen, Y., Amritkar, P., Thakur, R., Zhuang, Y.: A New Data Sieving Approach for High Performance I/O. In: Park, J.J.(J.H.), Leung, V.C.M., Wang, C.-L., Shon, T. (eds.) Future Information Technology, Application, and Service. LNNE, vol. 164, pp. 111–121. Springer, Heidelberg (2012)
9. Ching, A., Choudhary, A., Liao, W., Ross, R., Gropp, W.: Efficient Structured Data Access in Parallel File Systems. In: IEEE International Conference on Cluster Computing (2003)
10. Ching, A., Choudhary, A., Coloma, K., Liao, W.: Noncontiguous I/O Access Through MPI-IO. In: IEEE/ACM International Symposium on Cluster Computing and the Grid (2003)
11. Coloma, K., Ching, A., Choudhary, A., Liao, W., Ross, R., Thakur, R., Ward, L.: A New Flexible MPI Collective I/O Implementation. In: IEEE International Conference on Cluster Computing (2006)
12. Cha, K., Maeng, S.: An Efficient I/O Aggregator Assignment Scheme for Collective I/O Considering Processor Affinity. In: 7th International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems (2011)
13. Chai, L., Gao, Q., Panda, D.: Understanding the Impact of Multi-core Architecture in Cluster Computing. In: The 7th IEEE International Symposium on Cluster Computing and the Grid (2007)
14. Chaarawi, M., Gabriel, E.: Automatically Selecting the Number of Aggregators for Collective I/O Operations. In: IEEE International Conference on Cluster Computing (2011)
15. FLASH IO Benchmark Routine-Parallel HDF 5,
http://www.ucolick.org/zingle/flash_benchmark_io/
16. FT: Discrete 3D Fast Fourier Transform,
<http://www.nas.nasa.gov/publications/npb.html>
17. HDF5 documents, <http://www.hdfgroup.org/HDF5/whatishdf5.html>
18. Johnsson, S.L.: CMSSL: a Scalable Scientific Software Library. In: Proceedings of Scalable Parallel Libraries Conference, Mississippi State, MS (1993)
19. PVFS2, Parallel Virtual File System (Version 2), <http://www.pvfs.org/>

20. Riley, K.: Introduction to Flash,
[http://flash.uchicago.edu/site/flashcode/
user_support/tutorial_talks/home.py?submit=May2004.txt](http://flash.uchicago.edu/site/flashcode/user_support/tutorial_talks/home.py?submit=May2004.txt)
21. Tuning I/O Performance (2012),
[http://doc.opensuse.org/products/draft/SLES/
SLES-tuning_sd_draft/cha.tuning.io.html](http://doc.opensuse.org/products/draft/SLES/SLES-tuning_sd_draft/cha.tuning.io.html)
22. Thakur, R., Gropp, W., Lusk, E.: Data Sieving and Collective I/O in ROMIO. In: Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation, Annapolis, MD (1999)
23. Latham, R., Ross, R.: PVFS, ROMIO, and the noncontig Benchmark (2005)
24. Liao, W., Choudhary, A.: Dynamically Adapting File Domain Partitioning Methods for Collective I/O Based on Underlying Parallel File System Locking Protocols. In: ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (2008)
25. Lofstead, J., Klasky, S., Schwan, K., Podhorszki, N., Jin, C.: Flexible io and integration for scientific codes through the adaptable io system (adios). In: Proc. CLADE 2008 (2008)
26. Lustre File System (2012), <https://www.xyratech.com/products/lustre>
27. MPICH2, A High Performance Message Passing Interface,
<http://www.mcs.anl.gov/research/projects/mpich2/>
28. Madduri, K., Ibrahim, K., Williams, S., Im, E., Ethier, S., Shalf, J., Oliker, L.: Gyrokinetic Toroidal Simulations on Leading Multi- and Manycore HPC Systems. In: ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (2011)
29. Noncontig I/O Benchmark,
<http://www-unix.mcs.anl.gov/thakur/pio-benchmarks.html>
30. NAS parallel benchmarks, NASA Ames Research Center (2009),
<http://www.nas.nasa.gov/Software/NPB>
31. Spider - the Center-Wide Lustre File System,
[http://www.olcf.ornl.gov/kb_articles/
spider-the-center-wide-lustre-file-system/](http://www.olcf.ornl.gov/kb_articles/spider-the-center-wide-lustre-file-system/)
32. Zhang, C., Yuan, X., Srinivasan, A.: Processor Affinity and MPI Performance on SMP-CMP Clusters. In: 11th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (2010)
33. Zheng, G.: Achieving High Performance on Extremely Large Parallel Machines: Performance Prediction and Load Balancing. PhD Thesis (2005)
34. Zou, H., Sun, X., Ma, S., Duan, X.: A Source-Aware Interrupt Scheduling for Modern Parallel I/O Systems. In: 26th IEEE International Parallel & Distributed Processing Symposium (2012)
35. Zhang, X., Jiang, S., Davis, K.: Making Resonance a Common Case: A High-Performance Implementation of Collective I/O on Parallel File Systems. In: 23th IEEE International Parallel & Distributed Processing Symposium (2009)
36. Zhang, X., Xu, Y., Jiang, S.: YouChoose: A Performance Interface Enabling Convenient and Efficient QoS Support for Consolidated Storage Systems. In: 27th IEEE Symposium on Massive Storage Systems and Technologies (2011)

Fast and Energy-efficient Breadth-First Search on a Single NUMA System

Yuichiro Yasui¹, Katsuki Fujisawa¹, and Yukinori Sato²

¹ Kyushu University, Fukuoka, Japan

{y-yasui,fujisawa}@imi.kyushu-u.ac.jp

² JAIST, Ishikawa, Japan

yukinori@jaist.ac.jp

Abstract. Breadth-first search (BFS) is an important graph analysis kernel. The Graph500 benchmark measures a computer’s BFS performance using the traversed edges per second (TEPS) ratio. Our previous nonuniform memory access (NUMA)-optimized BFS reduced memory accesses to remote RAM on a NUMA architecture system; its performance was 11 GTEPS (giga TEPS) on a 4-way Intel Xeon E5-4640 system. Herein, we investigated the computational complexity of the bottom-up, a major bottleneck in NUMA-optimized BFS. We clarify the relationship between vertex out-degree and bottom-up performance. In November 2013, our new implementation achieved a Graph500 benchmark performance of 37.66 GTEPS (fastest for a single node) on an SGI Altix UV1000 (one-rack) and 31.65 GTEPS (fastest for a single server) on a 4-way Intel Xeon E5-4650 system. Furthermore, we achieved the highest Green Graph500 performance of 153.17 MTEPS/W (mega TEPS per watt) on an Xperia-A SO-04E with a Qualcomm Snapdragon S4 Pro APQ8064.

Keywords: Breadth-first search, graph algorithms, parallel algorithms, multicore processing.

1 Introduction

The breadth-first search (BFS) is one of the most important graph analysis kernels. It can be used to obtain some properties of the connections between the nodes in a given graph. BFS can be used as not only a standalone kernel but also a subroutine in several applications such as connected components [2], maximum flow [1], centrality [3, 4], and clustering [5]. In particular, some recent streams have represented mathematical graphs over wide areas, and these kernels have been used for further analysis^{1,2}. BFS has a linear theoretical complexity of $O(n + m)$ for a problem with $n = |V|$ vertices and $m = |E|$ edges in a given

¹ Stanford Network Analysis Project: <http://snap.stanford.edu>

² Human Brain Project: <http://www.humanbrainproject.eu>

graph $G = (V, E)$. This complexity is optimal for sequential computation; however, it is not efficient for parallel computation. Therefore, previous studies have proposed efficient parallel computation algorithms based on a *level-synchronized parallel BFS* for multicore single-node systems [12–16] and multicore multi-node systems [17–21]. Table 1 shows the BFS performance of each of these implementations in terms of the traversed edges per second (TEPS) ratio and TEPS ratio per watt. Our previous algorithm [16], which is based on Beamer’s algorithm [15], achieves a performance of 11.2 GTEPS. Our algorithm was adapted to a nonuniform memory access (NUMA) architecture by efficiently managing computer resources, as a result of which its TEPS ratio was higher than that of other algorithms. In this study, we propose a fast and highly energy-efficient parallel BFS algorithm for a NUMA system that incorporates additional speedup techniques to achieve a performance of over 30 GTEPS, which is around three times faster than our previous algorithm. The primary contributions of this study are as follows:

1. An investigation of the major bottleneck in our NUMA-optimized BFS on a NUMA system.
2. A fast and highly energy-efficient BFS algorithm that applies degree-based speedup techniques on a NUMA system.

Table 1. BFS performance (TEPS ratio and TEPS/W) in related work

Authors	Base architecture (#CPU cores)	$\log_2 n$	m/n	GTEPS	MTEPS/W
Reference code [8]	4-way Intel Xeon E5-4640 (64)	27	16	0.1	0.20
Madduri et al. [12]	Cray MTA-2 (40)	21	512	0.5	-
Agarwal et al. [13]	4-way Intel Xeon 7500 (64)	20	64	1.3	-
Beamer et al. [14]	4-way Intel Xeon E7-8870 (80)	28	16	5.1	-
Yasui et al. [16]	4-way Intel Xeon E5-4640 (64)	26	16	11.2	17.39
This study	4-way Intel Xeon E5-4640 (64)	27	16	29.0	45.43
	4-way Intel Xeon E5-4650 (64)	27	16	31.7	41.01
	SGI Altix UV 1000 (512)	30	16	37.7	1.89
	Sony Xperia-A-SO-04E (4)	20	16	0.48	153.17
	ASUS Nexus 7 –2013 version– (4)	20	16	0.53	129.63

2 Brief Introduction to BFS and Related Work

2.1 BFS, Graph500, and Green Graph500

Breadth-First Search. We assume that the input of a BFS is a graph $G = (V, E)$ consisting of a set of vertices V and a set of edges E . The connections of G are contained as pairs (v, w) , where $v, w \in V$. The set of edges E corresponds to a set of adjacency lists A , where an adjacency list $A(v)$ contains the adjacency vertices w of outgoing edges $(v, w) \in E$ for each vertex $v \in V$. A BFS explores the various edges spanning all other vertices $v \in V \setminus \{s\}$ from the source vertex $s \in V$ in a given graph G and outputs the *predecessor map* π , which is a map from each vertex v to its parent. When the predecessor map $\pi(v)$ points to only one parent for each vertex $v \in V$, it represents a tree with the root vertex $s \in V$.

Graph500 and Green Graph500. The Graph500 benchmark³ is designed to measure the computer performance for applications that require an irregular memory access pattern. Following its announcement in June 2010, the first Graph500 list was released in November 2010. This benchmark must perform the following steps:

1. **Generation.** This step generates the edge list of the Kronecker graph [10] using a recursive matrix (R-MAT) computation [11]. This generator requires the scale and the edgefactor as inputs. It outputs the 2^{scale} vertices and $2^{\text{scale}} \cdot \text{edgefactor}$ edges that contain some self-loops and some parallel edges.
2. **Construction (timed).** This step constructs the graph representation from the edge list obtained in Step 1.
3. **BFS iterations (timed).** This step iterates the timed *BFS*-phase and the untimed *verify*-phase 64 times. The former executes the BFS for each source, and the latter confirms the output of the BFS.

Graph500 benchmark is based on the TEPS ratio, which is computed for a given graph and the BFS output [8]. The Green Graph500 benchmark⁴ is designed to measure the energy efficiency of a computer by using the TEPS ratio per watt [9]. These lists have been updated biannually since their introduction.

2.2 CPU Affinity and Local Memory Allocation on NUMA System

We provide some definitions of processor affinity and memory policy for NUMA architecture processors. The mapping of CPU cores is represented by the processor ID, package ID, core ID, and SMT ID. The processor ID is a unique integer associated with the processing element, and it is used to bind threads to cores by invoking the system call `sched_setaffinity()`. The package ID is a unique integer associated with the physical chip, and it is used to bind data to the memory by invoking the system call `mbind()`. Finally, the core ID and the SMT ID are unique integers associated with the processing element in the physical chip and the processing thread in the processing element, respectively.

In NUMA architecture systems, the memory access time depends on the memory location relative to a processor. A processor can access its local memory faster than it can its remote (nonlocal) memory (i.e., memory local to another processor or memory shared between processors). In this study, we propose general techniques for processor affinity and memory binding for NUMA architecture systems. We have already applied similar techniques in our previous work [16] as well as in graph algorithms for the shortest path problem [6] and mathematical optimization problem [7]. We have implemented the ubiquity library for intelligently binding cores (ULIBC) [16] for an automatic configuration system for CPU affinity and memory binding on a NUMA architecture and Linux system, as shown in Fig. 1. Our library provides some APIs in the parallel region, such as `get_numa_procid()` and `get_numa_cputopo()`, which require the thread ID

³ Graph500 benchmark: <http://www.graph500.org>

⁴ Green Graph500 benchmark: <http://green.graph500.org>

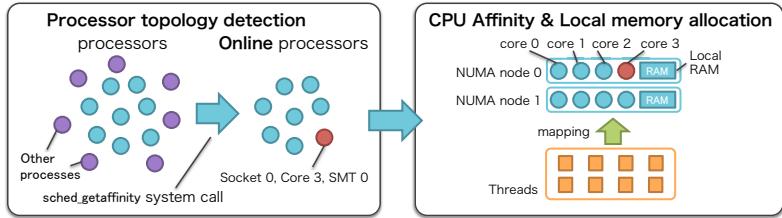


Fig. 1. Configuration of CPU affinity and local memory binding

and processor ID as inputs and return the processor ID for the Thread–CPU mapping and a tuple of (SMT ID, core ID, package ID) for the CPU topology, respectively.

2.3 NUMA-Optimized Hybrid BFS

Beamer et al. [14, 15] proposed a hybrid framework for the BFS algorithm (Algorithm 1) that reduced the number of edges explored. This algorithm combines two different traversal kernels: *top-down* and *bottom-up*. The former traverses *neighbors* Q^N of the *frontier* Q^F , whereas the latter finds the *frontier* from vertices in candidate neighbors (all unvisited vertices). This algorithm requires traversal (lines 6–9) and swaps Q^N and Q^F (line 10) at each level.

Algorithm 1. Hybrid BFS algorithm proposed by Beamer et al.

Input : directed graph $G = (V, A^F, A^B)$ and source vertex $s \in V$.
Data : frontier queue Q^F , neighbor queue Q^N , and visited flag *visited*.
Output: predecessor map of BFS tree $\pi(v)$.

```

1  $\pi(v) \leftarrow -1, \forall v \in V$ 
2  $visited \leftarrow \{s\}$ 
3  $Q^F \leftarrow \{s\}$ 
4  $Q^N \leftarrow \emptyset$ 
5 while  $Q^F \neq \emptyset$  do
6   if is_TopDown( $Q^F, Q^N, visited$ ) then                                /* Traversal */
7      $Q^N \leftarrow \text{Top-down}(G, Q^F, visited, \pi)$ 
8   else
9      $Q^N \leftarrow \text{Bottom-up}(G, Q^F, visited, \pi)$                                 /* Traversal */
10     $\text{swap}(Q^F, Q^N)$                                                  /* Swap */

```

Table 2 shows how a traversal policy is determined for *top-down* and *bottom-up* kernels (line 6). The traversal policy of the hybrid algorithm moves from *top-down* to *bottom-up* in the *growing* phase $|Q^F| < |Q^N|$, and it returns from *bottom-up* to *top-down* in the *shrinking* phase $|Q^F| \geq |Q^N|$. The algorithm uses the exact and approximate number of traversed edges m_F and m'_F in *top-down* and the approximate traversed edges of the *bottom-up*, m'_B , which are defined as follows:

$$m_{\mathcal{F}} \leftarrow |\{(v, w) \in E \mid v \in Q^N, w \in A^{\mathcal{F}}(v)\}|, \quad (1)$$

$$m'_{\mathcal{F}} \leftarrow |Q^N| \cdot \text{edgefactor}, \quad (2)$$

$$m'_{\mathcal{B}} \leftarrow |V \setminus \text{visited}| \cdot \text{edgefactor} + |V|. \quad (3)$$

In addition, we determine the optimum values of the switching parameters α and β , which are set as 16 and 16 based on the actual execution time.

Table 2. Traversal policy selection

(a) Growing phase $ Q^F < Q^N $			(b) Shrinking phase $ Q^F \geq Q^N $		
Current \ Next	Top-down	Bottom-up	Current \ Next	Top-down	Bottom-up
Top-down	$m_{\mathcal{F}} \cdot \alpha < m'_{\mathcal{B}}$	$m_{\mathcal{F}} \cdot \alpha \geq m'_{\mathcal{B}}$	Top-down	$m'_{\mathcal{F}} \cdot \beta < m'_{\mathcal{B}}$	$m'_{\mathcal{F}} \cdot \beta \geq m'_{\mathcal{B}}$
Bottom-up	—	always	Bottom-up		

Our NUMA-optimized algorithm, which is based on Beamer et al.'s hybrid algorithm, requires a given graph and working variables for a BFS to be divided into the local memory before the traversal. In our algorithm, the traversal phase avoids accessing the remote memory by using the following column-wise partitioning:

$$V = [V_0 \mid V_1 \mid \cdots \mid V_{\ell-1}], \quad A = [A_0 \mid A_1 \mid \cdots \mid A_{\ell-1}], \quad (4)$$

and each set of partial vertices V_k on k -th NUMA node is defined by

$$V_k = \left\{ v_j \in V \mid j \in \left[\frac{kn}{\ell}, \frac{k(n+1)}{\ell} \right) \right\}, \quad (5)$$

where n is the number of vertices and the divisor ℓ is set to the number of CPU sockets. In addition, to avoid accessing the remote memory, we define partial adjacency lists $A_k^{\mathcal{F}}$ and $A_k^{\mathcal{B}}$ for the *top-down* and *bottom-up* kernels as follows:

$$A_k^{\mathcal{F}}(v) = \{w \in \{V_k \cap A(v)\}\}, v \in V, \quad A_k^{\mathcal{B}}(w) = \{v \in A(w)\}, w \in V_k. \quad (6)$$

Furthermore, the working spaces Q_k^N , visited_k , and π_k for partial vertices V_k are allocated to the local memory on k -th NUMA node with memory pinned. Algorithms 2 and 3 and Figs 2(a) and 2(b) describe the *top-down* and *bottom-up* kernels for the NUMA-optimized BFS algorithm. Both algorithms bind the threads T_k to processors and local memory at k -th NUMA node without accessing the remote memory. Their respective computational complexities are $O(m)$ and $O(m \cdot \text{diam}_G)$, where m is the number of edges and diam_G is the diameter of the given graph; this is the same complexity as that of Beamer et al.'s previous hybrid algorithm. Therefore, the hybrid algorithm that combines these algorithms has $O(m \cdot \text{diam}_G)$ complexity. However, the actual CPU time of the

hybrid algorithm is shorter than that of top-down only for a small-world network such as a Kronecker graph.

Algorithm 2. NUMA-optimized top-down BFS.

Input : number of CPU sockets ℓ , NUMA node IDs
 $k = \{0, 1, \dots, \ell - 1\}$, directed graph $G = \{G_k\} = \{(V_k, A_k^F)\}$,
frontier queue (local copy) Q^F , visited flag $visited = \{visited_k\}$,
and predecessor map of BFS tree $\pi = \{\pi_k\}$.
Output: neighbor queue $Q^N = \{Q_k^N\}$.

```

1  $T_k \leftarrow \emptyset, \forall k \in \ell$ 
2 fork()
3  $t \leftarrow \text{omp\_get\_thread\_num}()$ 
4 sched_cpaaffinity(get_numa_procid( $t$ ))
5  $(i, j, k) \leftarrow \text{get\_numa\_cputopo}(\text{get\_numa\_procid}(t))$ 
6  $T_k \leftarrow T_k \cup \{(i, j)\}$ 
7  $Q_k^N \leftarrow \emptyset$ 
8 for  $v \in Q^F$  in parallel( $T_k$ ) do
9   for  $w \in A_k^F(v)$  do
10    if  $w \notin visited_k$  atomic then
11       $\pi_k(w) \leftarrow v$ 
12       $visited_k \leftarrow visited_k \cup \{w\}$ 
13       $Q_k^N \leftarrow Q_k^N \cup \{w\}$ 
14 join()

```

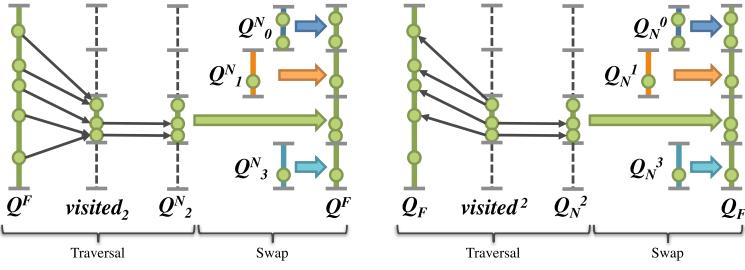
Algorithm 3. NUMA-optimized bottom-up BFS.

Input : number of CPU sockets ℓ , NUMA node IDs
 $k = \{0, 1, \dots, \ell - 1\}$, directed graph $G = \{G_k\} = \{(V_k, A_k^F)\}$,
frontier queue (local copy) Q^F , visited flag $visited = \{visited_k\}$,
and predecessor map of BFS tree $\pi = \{\pi_k\}$.
Output: $Q^N = \{Q_k^N\}$: neighbor queue.

```

1  $T_k \leftarrow \emptyset, \forall k \in \ell$ 
2 fork()
3  $t \leftarrow \text{omp\_get\_thread\_num}()$ 
4 sched_cpaaffinity(get_numa_procid( $t$ ))
5  $(i, j, k) \leftarrow \text{get\_numa\_cputopo}(\text{get\_numa\_procid}(t))$ 
6  $T_k \leftarrow T_k \cup \{(i, j)\}$ 
7  $Q_k^N \leftarrow \emptyset$ 
8 for  $w \in V_k \setminus visited_k$  in parallel( $T_k$ ) do
9   for  $v \in A_k^B(w)$  do
10    if  $v \in Q^F$  then
11       $\pi_k(w) \leftarrow v$ 
12       $visited_k \leftarrow visited_k \cup \{w\}$ 
13       $Q_k^N \leftarrow Q_k^N \cup \{w\}$ 
14      break
15 join()

```



(a) Top-down on 2-th NUMA node. (b) Bottom-up on 2-th NUMA node.

Fig. 2. Overview of NUMA-optimized BFS on ℓ -way NUMA system ($\ell = 4$)

3 Degree-Aware BFS

3.1 Bottleneck Analysis of Hybrid-BFS

In this section, we explain our proposed *Degree-aware BFS* algorithm that improves the major bottleneck in NUMA-optimized BFS. First, we show the major bottleneck in NUMA-optimized BFS, namely, the bottom-up kernel. The bottom-up step checks that each unvisited vertex has an adjacent vertex that connects vertices in the frontier. Therefore, this step reduces unnecessary edge traversals if an adjacency vertex that has already been visited with high probability is allocated at a higher position of each adjacency vertex list. It is difficult to obtain the optimal ordering for the adjacency vertex list. however, we focus on the out-degree $\deg_G(v)$ of each vertex $v \in V$, which is defined as follows:

$$\deg_G(v) = |A(v)|, v \in V. \quad (7)$$

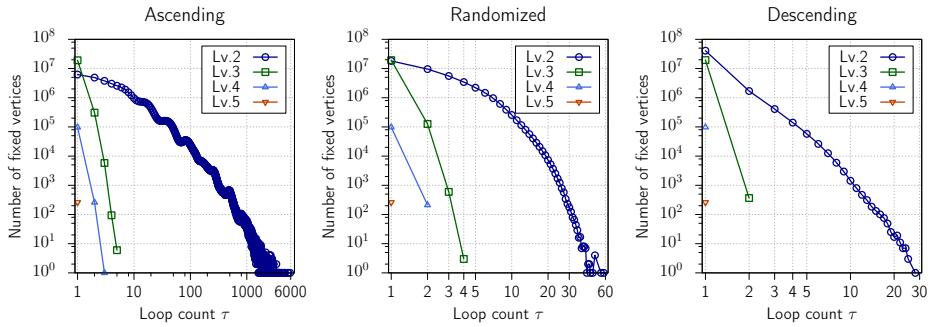
Table 3 shows a comparison of the number of traversed edges for each level in *top-down* and *bottom-up*, and each ordering, such as *Ascending*, *Randomized*, and *Descending*. *Ascending* and *Descending* construct an adjacency vertex list $A(v)$ that is sorted by $\deg_G(w), w \in A(v)$ in ascending and Descending order, respectively. *Randomized* constructs an adjacency vertex list $A(v)$ that is randomized. The table shows that most traversed edges were concentrated in Level-2 and that the number of traversed edges is affected by each ordering.

Fig. 3 shows the distribution of the loop count τ of the bottom-up step in each level. These figures can be used to easily understand the properties of first adjacency lists. The maximum count of the bottom-up loop ($\max \tau = 28$) using descending order is much smaller than that ($\max \tau = 5,873$) using ascending order. This clearly suggests that the ordering strategies affect the loop count.

Finally, we investigate the breakdown of processing in each vertex for each adjacency ordering shown in Table 4. The table shows that the number of zero-degree vertices is half the total number of vertices. These orderings have the same property in that most traversals of unvisited vertices occur in bottom-up searching. In particular, when using descending order, a BFS requires the most total CPU time in bottom-up at the first loop ($\tau = 1$).

Table 3. Number of traversed edges in a BFS for Kronecker graph with scale 27

Level	Top-down	Bottom-up	Hybrid algorithm		
			Step	Ascending	Randomized
0	22	4,223,250,243	T	22	22
1	239,930	3,258,645,723	T	239,930	239,930
2	1,040,268,126	83,878,899	B	848,743,124	150,006,673
3	3,145,608,885	19,616,130	B	19,935,737	19,742,764
4	37,007,608	139,606	B	139,868	139,817
5	98,339	41,846	B	41,846	41,846
6	260	41,586	T	260	260
Total	4,223,223,170	7,585,614,033	–	869,100,787	170,171,312
%	100 %	179.6 %		20.6 %	4.0 %
					2.5 %

**Fig. 3.** Distribution of the loop count τ of the bottom-up step at each level in a BFS for Kronecker graph with SCALE27 and edgefactor 16**Table 4.** Breakdown of bottleneck in a BFS of a Kronecker graph with scale 27

(a) Ascending order				
Component	0-degree	Bottom-up ($\tau = 1$)	Bottom-up ($\tau \geq 2$)	Top-down
#vertices	71,140,085	25,489,401	37,331,644	215,070
Ratio	53.00 %	18.99 %	27.81 %	0.16 %
(b) Randomized order				
Component	0-degree	Bottom-up ($\tau = 1$)	Bottom-up ($\tau \geq 2$)	Top-down
#vertices	71,140,085	37,663,087	25,157,958	215,070
Ratio	53.00 %	28.06 %	18.74 %	0.16 %
(c) Descending order				
Component	0-degree	Bottom-up ($\tau = 1$)	Bottom-up ($\tau \geq 2$)	Top-down
#vertices	71,140,085	60,462,127	2,358,918	215,070
Ratio	53.00 %	45.05 %	1.76 %	0.16 %

3.2 Degree-Aware BFS on Degree-Aware Graph Representation

The reference to the adjacent vertices require many in-directing accesses via an index array and an adjacency edge array of the compressed sparse row (CSR) format graph. However, we clarified that the major bottleneck of the hybrid BFS

algorithm is the edge traversal of the first adjacent vertex in the bottom-up step in the previous subsection. Then, we separated the standard CSR graph into the *highest-degree adjacency vertex list* $A^{\mathcal{B}+}$ and resting CSR graph $A^{\mathcal{B}-}$. The highest-degree adjacency vertex $A^{\mathcal{B}+}(v)$ for each vertex v contains an adjacency vertex w , whose maximum degree is given as follows:

$$A^{\mathcal{B}+}(v) = \arg \max_{w \in A^{\mathcal{B}}(v)} \{ |d_G(w)| \}, v \in V. \quad (8)$$

This graph representation requires additional computational overhead only for sorting the adjacency vertex list, and it does not suffer from the problem of requiring increasing memory. We focus on the fact that half of the total number of vertices are zero-degree vertices. This property does not have much effect on the performance of the top-down search. However, the bottom-up search is affected because the frontier is searched from all unvisited vertices, including zero- and nonzero-degree vertices. To avoid the access cost of zero-degree vertices, we propose *zero-degree vertex suppression*, which renames the vertex ID of only each nonzero vertex during graph construction. Algorithm 4 describes the degree-aware bottom-up BFS. It uses a graph representation that separates the highest-degree adjacency vertex $A^{\mathcal{B}+}$ and the resting adjacency vertices $A^{\mathcal{B}-}$. This algorithm separates two major loops for this purpose, such as lines 8–13 and lines 14–20.

We compared our speedup techniques—“highest-degree adjacency vertex list (high-deg)” and “zero-degree suppression (zero-deg)”—for a Kronecker graph with scale 27 on a SandyBridge-EP system, the results of which are shown in Table 5. Zero-deg showed two times improved performance relative to our previous NUMA-optimized BFS. In comparison, high-deg showed little effect (a speedup of about 34%) on the BFS performance. However, a combination of these two methods showed 2.68 times faster performance.

Table 5. Comparison of our speedup techniques on Sandybridge-EP

Implementation	SCALE	GTEPPS	Speedup
NUMA-opt. [16]	27	10.85	× 1.00
NUMA-opt. + High-deg	27	14.55	× 1.34
NUMA-opt. + Zero-deg	27	22.01	× 2.03
NUMA-opt. + High-deg + Zero-deg	27	29.03	× 2.68

Table 6. Machine environments

Machine	Processor	$p = \ell \times t$	RAM	LLC	CC
Westmere-EP	Intel Xeon E7-4870 2.93GHz	24 = 2 × 12	96 GB	12 MB	gcc-4.4
Magny-Cours	AMD Opteron 6174 2.20GHz	48 = 8 × 6	256 GB	512 KB	gcc-4.8
Westmere-EX	Intel Xeon E7-4870 2.40GHz	80 = 4 × 20	512 GB	30 MB	gcc-4.4
SandyBridge-EP	Intel Xeon E5-4640 2.40GHz	64 = 4 × 16	512 GB	20 MB	gcc-4.4
SandyBridge-EP (2.7GHz)	Intel Xeon E5-4650 2.70GHz	64 = 4 × 16	512 GB	20 MB	gcc-4.4
Altix UV1000	Intel Xeon E7-8837 2.67GHz	512 = 64 × 8	4096 GB	24 MB	icc-12.1

Algorithm 4. Degree-aware NUMA-optimized bottom-up BFS.

Input : number of CPU sockets ℓ , NUMA node IDs $k = \{0, 1, \dots, \ell - 1\}$, directed graph $G = \{G_k\} = \{(V_k, A_k^{\mathcal{B}+}, A_k^{\mathcal{B}-})\}$, frontier queue (local copy) Q^F , visited flag $visited = \{visited_k\}$, and predecessor map of BFS tree $\pi = \{\pi_k\}$.

Output: neighbor queue $Q^N = \{Q_k^N\}$.

```

1  $T_k \leftarrow \emptyset, \forall k \in \ell$ 
2 fork()
3  $t \leftarrow \text{omp\_get\_thread\_num}()$ 
4 sched_cpaaffinity(get_numa_cputopo(get_numa_procid(t)))
5  $(i, j, k) \leftarrow \text{get\_numa\_cputopo}(\text{get\_numa\_procid}(t))$ 
6  $T_k \leftarrow T_k \cup \{(i, j)\}$ 
7  $Q_k^N \leftarrow \emptyset$ 
8 for  $w \in V_k \setminus visited_k$  parallel( $T_k$ ) do
9    $v \leftarrow A_k^{\mathcal{B}+}(w)$ 
10  if  $v \in Q^F$  then
11     $\pi_k(w) \leftarrow v$ 
12     $visited_k \leftarrow visited_k \cup \{w\}$ 
13     $Q_k^N \leftarrow Q_k^N \cup \{w\}$ 
14 for  $w \in V_k \setminus visited_k$  parallel( $T_k$ ) do
15   for  $v \in A_k^{\mathcal{B}-}(w)$  do
16     if  $v \in Q^F$  then
17        $\pi_k(w) \leftarrow v$ 
18        $visited_k \leftarrow visited_k \cup \{w\}$ 
19        $Q_k^N \leftarrow Q_k^N \cup \{w\}$ 
20       break
21 join()

```

4 Numerical Results

4.1 Implementation and Machine Environments

Table 6 shows the environments of each NUMA architecture system, such as the processor, RAM size, last level cache (LLC) size, and C compiler (CC). Each system has a maximum of p threads on ℓ -way NUMA nodes, each of which contains t logical cores. We used the optimization option setting of `-O2`.

4.2 BFS Performance on Graph500 Benchmark

Performance Variation with Problem Size. Table 7 shows a comparison of the performance of the reference code (Graph500 version 2.1.4), our previous algorithm (NUMA-optimized), and our algorithm (Degree-aware, This study) for a Kronecker graph with edgefactor 16 on the SandyBridge-EP. The speedup ratio of our algorithm with respect to the reference code generally increases with scale for the Kronecker graph. Each blank in the table indicates that the reference

code generated a serious error and was aborted. Fig. 4 shows a comparison of the performance of our BFS on different NUMA systems with respect to scale. However, our BFS causes performance degradation because the frontier queue size is larger than the last level cache size, as also reported in Agarwal et al. [13].

Table 7. Performance (GTEPS and speedup ratio) of the reference code and our algorithm (NUMA-optimized and Degree-aware) on SandyBridge-EP

SCALE	Reference	NUMA-optimized [16]	Degree-aware	
	GTEPS	GTEPS Speedup	GTEPS	Speedup
20	0.358	4.578 × 12.8	4.994	× 14.0
21	0.233	6.317 × 27.1	8.649	× 37.1
22	0.101	8.433 × 83.7	13.896	× 137.6
23	0.129	10.023 × 77.6	17.859	× 138.4
24	0.123	10.829 × 88.3	22.317	× 181.4
25	0.107	11.155 × 104.4	25.269	× 236.2
26	0.097	11.149 × 114.9	26.675	× 275.0
27	0.087	10.854 × 124.5	29.034	× 333.7
28	—	9.887 —	23.522	—
29	—	9.393 —	21.381	—

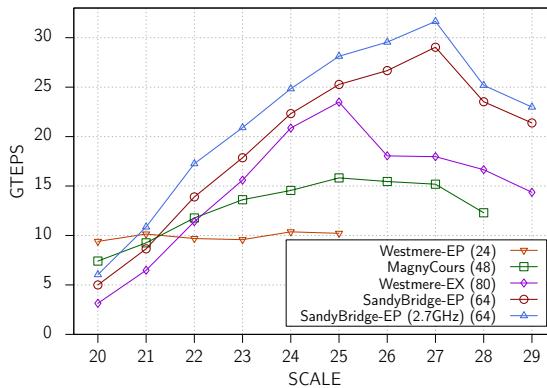
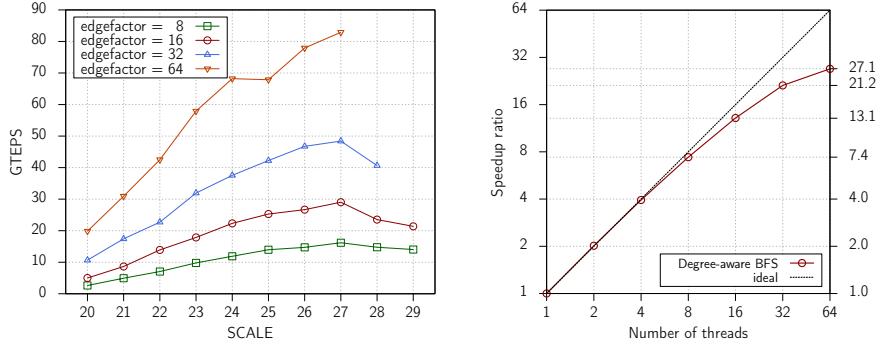


Fig. 4. Variation in BFS performance (GTEPS) with problem size on NUMA machines

Performance Variation with Edgefactor and Scalability. Fig. 5(a) shows how the performance of our algorithm varied with the scale and edgefactor of the Kronecker graph. The blanks in the figures indicate points at which the system had insufficient memory space to execute the algorithm. For the Kronecker graph with a scale 27, our BFS attained a peak TEPS for edgefactors of 8, 16, 32, and 64. Our BFS is more efficient for a Kronecker graph with a large edgefactor (64) than for one with a small edgefactor (8). This speedup afforded by *bottom-up* reduces the edge traversals required for a dense graph (large edgefactor). Furthermore, fig. 5(b) shows that our BFS with 64 threads is 27.1 times faster than sequential.



(a) Performance (GTEPS) varied with problem size and edgefactor (b) Strong scaling for SCALE 27 and edgefactor 16

Fig. 5. Performance (GTEPS) and scalability of our algorithm on SandyBridge-EP

4.3 BFS Performance on SGI Altix UV1000 System

We discuss the BFS performance on an SGI Altix UV1000, which actualizes massive thread parallel computing based on cache-coherent nonuniform memory access (ccNUMA). A rack of SGI Altix UV1000 systems contains 192 NUMA nodes with an Intel Xeon CPU E7-8837 2.67 GHz and 64 GB RAM and generates up to 8 threads in parallel with hyperthreading disabled. Our BFS requires edge traversal to be executed in local RAM (Algorithm 1, lines 6–9) and the frontier queue and the neighbor queue for the next level to be swapped in remote RAM (Algorithm 1, line 10). We do not consider the swap operation as a bottleneck because most CPU time is spent on the traversal operations on other NUMA systems with a few CPU sockets. Table 8 shows the CPU time and the variation of the ratio of traversals and swaps with the number of threads p set as 128 (one-fourth of a rack), 256 (one-half of a rack), and 512 (one rack) for Kronecker graph with scale 30. As the number of threads increases, the bottleneck moves from the traversal operation in local memory to the swap operation in remote memory. Therefore, we focus on the fast computation of the swap operation.

Table 8. TEPS, CPU time, and ratio of traversal and swap on SGI Altix UV1000

p	GTEPS	Traversal on Local RAM	Swap on Remote RAM
128	18.76	732.80 ms (80%)	182.10 ms (20%)
256	26.17	437.90 ms (67%)	218.00 ms (33%)
512	37.70	257.90 ms (57%)	196.60 ms (43%)

4.4 BFS Performance on Real-World Networks

We verify our BFS performance by using real-world networks on a Sandybridge-EP system. Table 9 shows the graph sizes, graph properties, and GTEPS using

our BFS for each network instance. Here, diam'_G indicates the approximation of the diameter on each network using the maximum hops for each vertex of 64 BFS iterations. USA-road-d and wiki-Talk have approximately the same edgefactor, but the latter shows 4–11 times faster than the former owing to its diameter being a thousand times smaller relatively. On the other hand, LiveJournal and twitter have approximately the same diameter, but the latter shows 3–5 times faster than the former owing to its edgefactor being 1.68 times larger relatively. In addition, twitter and friendster show similar BFS performances of approximately 10 GTEPS because they have similar edgefactor and similar diameters. Therefore, we verify whether our BFS is affected by using both the edgefactor and diameter of the network. From these numerical results, we could achieve high performance for large-scale small-world networks with a large edgefactor.

Table 9. BFS performance of real-world network on Sandybridge-EP system

Instance	Graph size		edgefactor m/n	Diameter	GTEPS				
	n	m			diam' _{G}	min	1/4	median	3/4
wiki-Talk [23, 24]	2.39 M	5.02 M	2.1	8	0.29	0.61	0.75	0.87	1.26
USA-road-d [25]	23.95 M	58.33 M	2.4	8,098	0.07	0.08	0.09	0.09	0.11
LiveJournal [26, 27]	4.85 M	68.99 M	14.2	16	2.76	3.76	4.07	4.32	4.94
twitter [28]	61.58 M	1,468.37 M	23.8	16	7.58	10.02	10.90	12.68	24.09
friendster [29]	65.61 M	1,806.07 M	27.5	25	4.89	9.61	10.74	11.29	11.81

5 Energy Efficiency of Our BFS

Thus far, we have discussed the BFS performance in terms of only the speed. This section discusses the BFS performance of our implementation in terms of energy efficiency. Our speedup techniques are aimed at not only fast computation but also energy efficiency, as described in [4].

Fig. 6 shows the performance (GTEPS) and energy-efficiency (MTEPS/W) of our Degree-aware BFS, our NUMA-optimized BFS, and the Graph500 reference code for each CPU affinity on the SandyBridge-EP. If the number of threads is the same, our Degree-aware BFS achieves a high TEPS and a high TEPS/W with a larger number of sockets. On the other hand, if the number of sockets is the same, our BFS achieves a high TEPS and a high TEPS/W with a larger number of threads.

As an energy-efficient machine environment, we use Android-based devices, which have highly energy-efficient ARM or Snapdragon processors. We develop a native binary code based on the C/C++ programming language by using the Android native development kit (NDK). The Android NDK supports multithreaded computation using the OpenMP library and the atomic functions of the GCC extension. It enables binary code to be transferred to the device and then be executed using Android developer tools. This developer environment was published on the Android Developers website³.

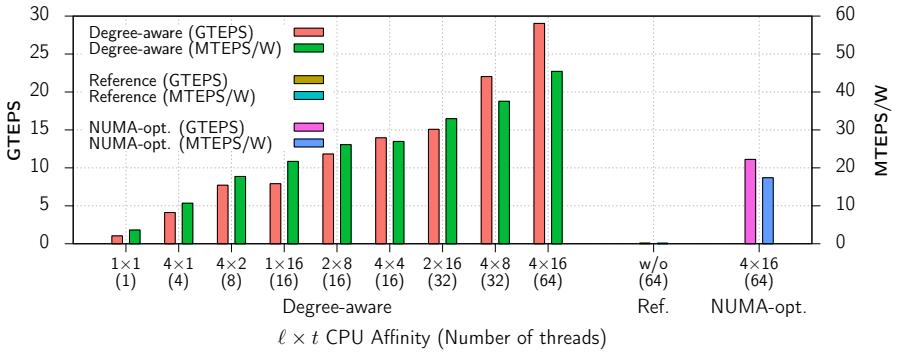


Fig. 6. Performance (GTEPS) and Energy-efficiency (MTEPS/W) of our Degree-aware BFS, our NUMA-optimized BFS, and Graph500 reference code for Kronecker graph with SCALE 27 on SandyBridge-EP with ℓ -NUMA-nodes \times t -threads CPU-affinity

Fig. 7 shows the performance of our degree-aware BFS and the reference BFS on Sony’s Android-based XperiaA SO-04E smartphone, which has a Qualcomm Snapdragon S4 Pro APQ8064 1.5 GHz processor and 2 GB RAM. Our BFS is assumed to be executed on a NUMA architecture system. It is difficult to achieve high performance directly using NUMA-optimized speedup techniques on a single CPU. However, we designed this algorithm to reduce unnecessary memory accesses and overheads of atomic operations, and as a result, we could achieve high performance even on Android devices, which do not have a NUMA architecture system. This figure shows that the reference code causes performance degradation. In comparison, our BFS achieves a maximum performance of 477.63 MTEPS for a Kronecker graph with scale 20 on XperiaA SO-04E with 4 threads.

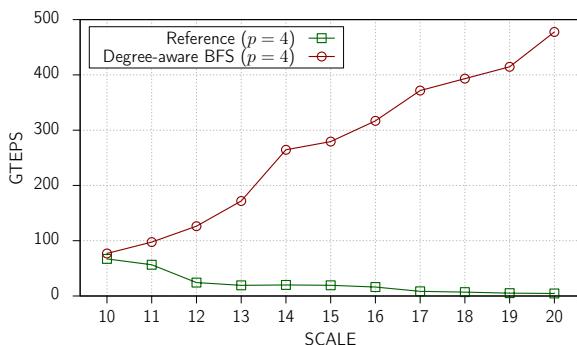


Fig. 7. MTEPS of reference BFS and Degree-aware BFS on XperiaA SO-04E

Table 10 summarizes the energy efficiency in terms of MTEPS, effective power (W), and MTEPS/W for BFS on a Kronecker graph with scale 20. The table shows that all algorithms have an effective power of around 3.0 W for BFS execution, suggesting that the effective power is not strongly affected by the number of threads and the algorithm used. With regard to energy-efficient computation, our degree-aware BFS is around 100 times faster than the reference code for roughly the same effective power of 3.0 W; specifically, our BFS shows an energy-efficient performance of 153.17 MTEPS/W.

Table 10. Energy efficiency of BFS for Kronecker graph with on XperiaA SO-04E

Implementation	SCALE	MTEPS	watt	MTEPS/W
Reference ($p = 1$)	20	3.25	3.15	1.03
Reference ($p = 4$)	20	4.58	3.22	1.42
Degree-aware ($p = 1$)	20	136.29	3.23	42.25
Degree-aware ($p = 2$)	20	248.08	2.99	82.92
Degree-aware ($p = 4$)	20	477.63	3.12	153.17

6 Conclusion

In this study, we investigate the major bottleneck in our previous NUMA-optimized BFS algorithm and apply degree-aware speedup techniques to it. Our new implementation achieved a BFS search performance of 31.65 GTEPS on 37.66 GTEPS on an SGI Altix UV1000 and a 4-way Intel Xeon E5-4650 system, which were ranked as the 50th (fastest on single node) and 51st (fastest on single server) best performances on the Graph500 list in November 2013, respectively. In addition, our BFS performance on a Sony Xperia-A-SO-04E and ASUS Nexus 7 (2013 version) was ranked as the first and second best performances in terms of energy efficiency on the Green Graph500 list in November 2013, respectively. Finally, further studies will be required to clarify the relationship between the theoretical complexity of our BFS and the small-world property.

Acknowledgments. We appreciate the valuable comments and suggestions from the anonymous reviewers. This research was supported by the Core Research for Evolutional Science and Technology (CREST) program of the Japan Science and Technology Agency (JST) and by the JAIST Research Center for Advanced Computing Infrastructure.

References

1. Edmonds, J., Karp, R.M.: Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM* 19(2), 248–264 (1972)
2. Cormen, T., Leiserson, C., Rivest, R.: *Introduction to Algorithms*. MIT Press, Cambridge (1990)
3. Brandes, U.: A Faster Algorithm for Betweenness Centrality. *J. Math. Sociol.* 25(2), 163–177 (2001)

4. Frasca, M., Madduri, K., Raghavan, P.: NUMA-Aware Graph Mining Techniques for Performance and Energy Efficiency. In: Proc. ACM/IEEE Int. Conf. High Performance Computing, Networking, Storage and Analysis (SC 2012), pp. 1–11. IEEE Computer Society (2012)
5. Girvan, M., Newman, M.E.J.: Community structure in social and biological networks. Proc. Natl. Acad. Sci. USA 99, 7821–7826 (2002)
6. Yasui, Y., Fujisawa, K., Goto, K., Kamiyama, N., Takamatsu, M.: NETAL: High-performance Implementation of Network Analysis Library Considering Computer Memory Hierarchy. J. Oper. Res. Soc. Japan 54(4), 259–280 (2011)
7. Fujisawa, K., Endo, T., Yasui, Y., Sato, H., Matsuzawa, N., Matsuoka, S., Waki, H.: Petascale General Solver for Semidefinite Programming Problems with Over Two Million Constraints. In: Proc. IEEE Int. Symp. Parallel and Distributed Processing (IPDPS 2014). IEEE Computer Society (2014)
8. Murphy, R.C., Wheeler, K.B., Barrett, B.W., Ang, J.A.: Introducing the Graph500. In: Cray User Group 2010 Proceedings (2010)
9. Hoefer, T.: GreenGraph500 Submission Rules,
<http://green.graph500.org/greengraph500rules.pdf>
10. Leskovec, J., Chakrabarti, D., Kleinberg, J., Faloutsos, C., Ghahramani, Z.: Kronecker Graphs: An Approach to Modeling Networks. J. Mach. Learning Res. 11, 985–1042 (2010)
11. Chakrabarti, D., Zhan, Y., Faloutsos, C.: R-MAT: A Recursive Model for Graph Mining. In: Proc. 4th SIAM Int. Conf. Data Mining, pp. 442–446. SIAM (2004)
12. Bader, D.A., Madduri, K.: Designing Multithreaded Algorithms for Breadth-First Search and st-connectivity on the Cray MTA-2. In: Proc. 2006 Int. Conf. Parallel Processing (ICPP 2006), pp. 523–530. IEEE Computer Society (2006)
13. Agarwal, V., Petrini, F., Pasetto, D., Bader, D.A.: Scalable Graph Exploration on Multicore Processors. In: Proc. ACM/IEEE Int. Conf. High Performance Computing, Networking, Storage and Analysis (SC 2010), pp. 1–11. IEEE Computer Society (2010)
14. Beamer, S., Asanović, K., Patterson, D.A.: Searching for a Parent Instead of Fighting Over Children: A Fast Breadth-first Search Implementation for Graph500. EECS Department, University of California, UCB/EECS-2011-117, Berkeley, CA (2011)
15. Beamer, S., Asanović, K., Patterson, D.A.: Direction-optimizing Breadth-first Search. In: Proc. ACM/IEEE Int. Conf. High Performance Computing, Networking, Storage and Analysis (SC 2012), p. 12. IEEE Computer Society (2012)
16. Yasui, Y., Fujisawa, K., Goto, K.: NUMA-optimized Parallel Breadth-first Search on Multicore Single-node System. In: Proc. IEEE Int. Conf. BigData 2013. IEEE Computer Society (2013)
17. Yoo, A., Chow, E., Henderson, K., McLendon, W., Hendrickson, B., Catalyürek, Ü.: A Scalable Distributed Parallel Breadth-first Search Algorithm on BlueGene/L. In: Proc. ACM/IEEE Conf. Supercomputing (SC 2005), p. 25. IEEE Computer Society (2005)
18. Buluç, A., Madduri, K.: Parallel Breadth-first Search on Distributed Memory Systems. In: Proc. ACM/IEEE Int. Conf. High Performance Computing, Networking, Storage and Analysis (SC11), p. 65. ACM (2011)
19. Petrini, F., Checconi, F., Willcock, J., Lumsdaine, A., Choudhury, A.R., Sabharwal, Y.: Breaking the Speed and Scalability Barriers for Graph Exploration on Distributed-memory Machines. In: Proc. ACM/IEEE Int. Conf. High Performance Computing, Networking, Storage and Analysis (SC 2012), p. 13. IEEE Computer Society (2012)

20. Ueno, K., Suzumura, T.: Highly Scalable Graph Search for the Graph500 Benchmark. In: Proc. 21st Int. ACM Symp. High-Performance Parallel and Distributed Computing (HPDC 2012), pp. 149–160. ACM (2012)
21. Ueno, K., Suzumura, T.: Parallel Distributed Breadth First Search on GPU. In: Proc. IEEE Int. Conf. High Performance Computing (HiPC 2013). IEEE Computer Society (2013)
22. McAuley, J., Leskovec, J.: Image Labeling on a Network: Using Social-Network Metadata for Image Classification. In: Fitzgibbon, A., Lazebnik, S., Perona, P., Sato, Y., Schmid, C. (eds.) ECCV 2012, Part IV. LNCS, vol. 7575, pp. 828–841. Springer, Heidelberg (2012)
23. Leskovec, J., Huttenlocher, D., Kleinberg, J.: Signed Networks in Social Media. In: CHI (2010)
24. Leskovec, J., Huttenlocher, D., Kleinberg, J.: Predicting Positive and Negative Links in Online Social Networks. In: WWW (2010)
25. The 9th DIMACS Implementation Challenge,
<http://www.dis.uniroma1.it/~challenge9/>
26. Backstrom, L., Huttenlocher, D., Kleinberg, J., Lan, X.: Group Formation in Large Social Networks: Membership, Growth, and Evolution. In: KDD (2006)
27. Leskovec, J., Lang, K., Dasgupta, A., Mahoney, M.: Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. Internet Mathematics 6(1), 29–123 (2009)
28. Kwak, H., Lee, C., Park, H., Moon, S.: What is Twitter, a Social Network or a News Media? In: Proceedings of the 19th International Conference on World Wide Web (WWW 2010), pp. 591–600 (2010)
29. Yang, J., Leskovec, J.: Defining and Evaluating Network Communities based on Ground-truth. In: ICDM (2012)

Evaluation of the Impact of Direct Warm-Water Cooling of the HPC Servers on the Data Center Ecosystem

Radosław Januszewski, Norbert Meyer, and Joanna Nowicka

Poznan Supercomputing and Networking Center, Poland
{radoslaw.januszewski,norbert.meyer}@man.poznan.pl,
joanna.e.nowicka@gmail.com

Abstract. The last 10 years we have witnessed a rapid growth of the computational performance of servers used by the scientific community. This trend was especially visible in the HPC scene, where the price per FLOPS decreased, while the packing density and power consumption of the servers increased. This, in turn changed significantly challenges and costs of keeping the environmental conditions. Currently operational costs, mainly the power bill, over the lifetime of a computing system overshadow the acquisition costs. In addition, the overheads on the consumed power introduced by the need of cooling the systems may be as big as 40%. This is a huge portion of the costs, therefore, optimizations in this area should be beneficial in terms of both economy and efficiency. There are many approaches for optimizations of the costs, mainly focusing on the air cooling. Contrary to these have we decided to scrutinize a different approach. We planned to use warm (up to 45 °C inlet temperature) as the cooling medium for computing cluster and check if using this way of cooling can introduce significant savings and, at the same time, we can simplify the cooling infrastructure making it more robust and energy efficient. Additionally, in our approach we tried to use variable coolant temperature and flow to take maximum advantage of so called free cooling, minimizing the power consumption of the server-cooling loop pair.

To validate the hypothesis PSNC (Poznan Supercomputing and Networking Center) built a customized prototype system which consists of hybrid CPU and GPU computing cluster, provided by the company Iceotope, along with a customized, highly manageable and instrumented cooling loop.

In the paper we analyze the results of using our warm-water liquid cooled system to see if and, if it is the case, what are the positive and negative consequences for the data center ecosystem.

Keywords: HPC, data center cooling, energy consumption, warm water cooling, direct liquid cooling.

1 Introduction and Motivation

Various research results show that energy is the key aspect that is influencing the TCO (Total Cost of Ownership) of the computing infrastructure[1]. While

the power consumed by the servers themselves is more than 50% of the Data Center, the growing power and heat density caused the bill for energy consumed by cooling infrastructure to sum up to 35-40% of the total energy costs of a data center. This is only the energy consumed by the Data Center infrastructure, one have to add a 10-15% of the energy consumed by the servers because that is amount of energy consumed by the servers fans[2]. One can see that by doing the cooling in a smart way may reduce the costs significantly. There are many papers focusing on optimization of the efficiency of the cooling, most of them however are focused on three topics: optimize airflow, increase the temperature inside the data center, use free cooling [3][4][5].

Typically the cooling systems in the Data Center rooms are designed to work within predefined parameter ranges. Traditionally this is justified by the need for constant environmental conditions for the servers as they are designed to work within certain temperature range that is usually quite narrow usually 20–25 °C[6] inlet air. To maintain this temperature a cold transfer medium, usually chilled water, has to be kept at around 15 °C. The costs of keeping the coolant at this temperature depends on various factors but most of all on the climate conditions in the area where the Data Center is built. In most cases the climate is cold enough for the cooling loop to operate in so called free cooling mode, where the chillers are by-passed and the cooling is relying only on the heat exchangers. This shows us that, in order to efficiently cool the Data Center, one have to either build it in a cold area or increase the coolant temperature.

Alternatively, one can increase the temperature in the water cooling loop, and thus increase the number of hours when the cooling system can operate bypassing the chillers. This solution is advocated by cooling infrastructure vendors that claim that increasing the temperature by 1 °C may introduce savings as big as 4%[7]. Unfortunately, this results in increased temperature of the air on the intake side of the servers. To keep thermal conditions of the electronics inside, the fans must operate at higher speeds, consuming additional energy. Depending on the climate one is operating its data center, this additional power draw adds more costs to the total power bill than one can save on the infrastructure costs.

In this paper we present a novel approach of using a direct liquid cooling solution, cooled by a customized cooling loop. Contrary to the traditional approach where the coolant is kept within a predefined temperature range, in our solution the coolant flow and temperature are driven by the climate conditions and the load on the cluster.

We are tackling the problem by changing the way the servers are cooled and by changing how the cooling loop is managed. The prototype system which consist of a cluster and a cooling loop pair was built as a part of the PRACE[8] project. In this system only the network switches are air cooled. The rest of the system: servers, storage, powers supplies are cooled directly with water. As for the cooling loop, instead of traditional fixed boundary temperatures, we are following the external weather conditions to keep the coolant temperature as low as possible, minimizing the power consumption of the loop at the same time.

2 Background and System Configuration

Currently direct water cooling seems to gain popularity but at the moment, when the prototype system was built, there were only few solutions on the market of x86 servers available. IBM iDataPlex was the only one delivered by one of the big vendors but, because in this product only CPUs and memory was cooled by water, this system was not able to fulfill the primary requirement: all components of the cluster should be directly liquid cooled. Therefore, PSNC decided to choose Iceotope[9] as the system vendor. The Iceotope standard solution was modified for the purpose of the project.

The computing cluster consists of 40 dual 6-core Xeon E5 servers working at 2.0 GHz (2.3 GHz on when turbo mode was enabled). In addition in the cluster we have installed 12 AMD S9000 GPGPU cards installed in 6 modules connected to the selected servers with external PCI connections. Each server is equipped with 32 or, in case of GPU serving ones, 64GB of memory, 2x1Gbit Ethernet, QDR InfiniBand interface and local SSD drive. In addition to the cluster, PSNC has ordered a customized, highly manageable and monitored cooling loop that was designed to provide maximum efficiency while being as simple as possible. The entire cooling path consisted of three steps:

The first one was an internal server cooling. Each server in the cluster is built as module filled with Novec liquid which acts as a heat conductor that transfers the heat from the chips to the water-Novec heat exchanger installed in the side of the module. All electronic parts (motherboard, CPUs, memory, SSD drives) are put inside the module and are completely submerged in the Novec. The server modules are installed vertically so the convection is ensuring proper Novec flow along the heat exchanger and the electronics.

The second step, later referenced as the primary loop (Figure 1), is the cooling loop inside the cluster rack that is transferring the heat from the server modules to the heat exchangers installed in the bottom of the server rack. In this loop we use demineralized water as the coolant. In the rack there are two semi-independent pumps working in parallel. This loop is working on negative pressure ensuring that eventual leaks result in the air being sucked in the loop rather than spilling the water out of the loop.

The third step, a secondary loop (Figure 1), is used to transfer the heat from the server rack inside data center room to a dry cooler unit placed on the roof of the building. The dry cooler is equipped with an adiabatic cooling support that should help keep the coolant within desired temperature range even during hot days. Because this loop is designed to operate also in winter conditions it is using a 30% glycol-water mixture as the coolant. All temperature sensors in this loop are installed in-flow therefore ensuring exact measurements.

The cooling loop control may work in one of three automatic modes. The first one tries to minimize the power consumed by the cooling loop by trying to operate in the high range of the servers specifications (45 °C inlet and 50 °C outlet temperatures). In the second mode the controller automatically adjusts the setting of the pump and valves to keep the inlet and outlet temperatures

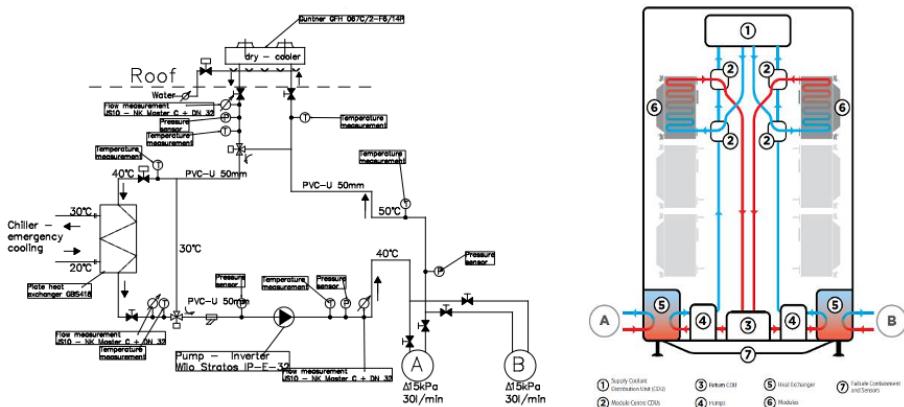


Fig. 1. Cooling loops: primary (right) and secondary (left)

within previously set temperature range. Alternatively there is a possibility of manual control over each of the elements (pumps, valves etc.) to allow for more sophisticated control algorithms. The first mode proved to be impractical because if the machine was idle the coolant flow in the secondary loop dropped to a level that was not acceptable by the servers.

During the tests presented in this paper we used the second mode, adjusting the inlet and outlet set points of the coolant going in and out of the cluster.

Although the cooling loop was designed to be able to handle the servers covering all external temperatures in Pozna, we decided to have a backup connection to the facility chilled water loop. Except from testing this loop was never used during presented tests or normal operations.

3 Testing Procedure

To assess the influence of the machine on the environment we put the maximum load on the servers by running single instance of the High Performance Linpack on 34 nodes and, separate HPL instance on 6 nodes and 12 GPU cards. The tests were repeated to produce constant load. The entire test procedure consisted of 5 cycles, after each cycle the inlet temperature to the machine was increased by 5 degrees starting from 15 °C inlet and ending with 35 °C inlet temperature. For each inlet temperature we have checked different deltas of the inlet-outlet temperatures. Original plan assumed starting with 2K delta and ending with 10 °C with 2K step, influencing the flow in the secondary loop. It turned out that having a constant 2K delta was not possible with fully loaded system, therefore the final tests were done with 3,4,6,8 and 10K deltas. The internal server loop is running at constant speed. Because of thermal inertia of the cluster, after each temperature change of the server inlet temperature, a 30 min period to allow the machine to stabilize the temperature. Both thermal and power sensors internally

were read every 5 seconds and then the values are averaged in 1 minute period and put in the database. The averaged values were used to produce the final results.

The tests are not covering the top end of the spectrum (inlet temperatures of 40 °C and more), the reasoning for that is explained in the following chapter.

Since the primary cooling loop is working at a constant pace, only the components of the secondary loop were controllable and allowed manipulations leading to energy optimizations. In this loop the most power hungry components were the pump and the fans of the rooftop heat exchanger. Because the fan activity is mostly affected by the external conditions and it is relatively low power (maximum 200W of power consumption) and is in a great degree dependent on the climate conditions, we have focused on optimization of the pump activity.

The pump was running at minimum of 50% because with this setting it was producing the minimum flow required by the cluster and, at the same time, minimum pressure required to transfer the coolant to the rooftop. By looking at the power consumption profile of the pump (Figure 2) one can see that the power consumption difference between 50% and 100% is almost an order of magnitude therefore it is worth keeping of the pump activity as low as possible.

Because the flow rate of the coolant via both rack and rooftop heat exchanger depend both on pump setting and the setting of the three way valve directing flow to the rooftop heat exchanger, data on Figure 2 was collected when the valve was fully opened.

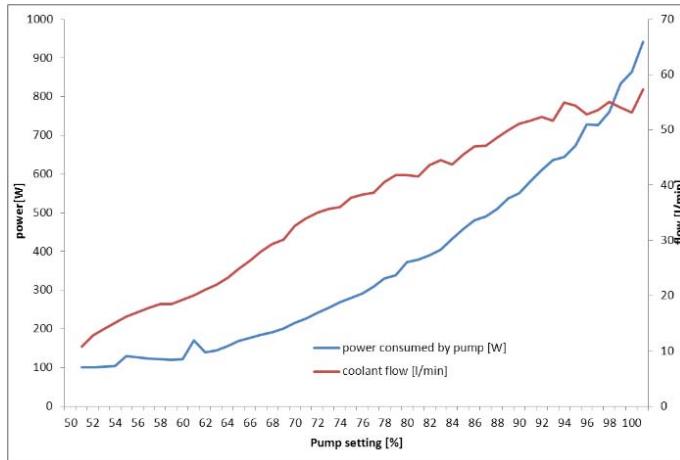


Fig. 2. Pump power consumption and coolant flow in relation to pump setting

During the rest of the tests the valve was adjusted automatically by the cooling loop control system.

All the historical data presented in this paper covers the period from the 1st February 2013 to the 1st February 2014.

4 Results

While increasing the temperature of the coolant, advocated by cooling equipment vendors, is undoubtedly beneficial from the cooling-loop energy efficiency point of view, as it allows to increase number of hours during the year when one can bypass the chillers and thus save the energy, it causes the rest of the ecosystem to consume more energy. First of all is the increased power consumption of the cluster. Both increasing the inlet temperatures and increasing the inlet-outlet delta, thus reducing the pump activity, is causing an increase of the temperature of the chips and therefore the energy consumed by the system. In case of our prototype, the maximum power draw difference between system cooled by 15 °C and 35 °C we were able to measure was 10%

Depending on the architecture, scale of the system, installed cooling system and climate the 10% of difference in maximum power consumption of the system between 15 °C and 35 °C inlet may in some cases overshadow the benefits gained from the reducing the power consumed by the cooling loop.

Modern CPUs are able to automatic increase of the clock depending on the thermal conditions and load put on the chip. In case of our test system all CPUs were automatically overclocked from base 2.0 GHz to 2.3 GHz. When the inlet temperature reached 40 °C, the some of the processors in the cluster started reaching thermal threshold and returned to base clock. Therefore we decided to stop the tests at 35 °C inlet temperature to keep the performance and the power consumption of the cluster consistent.

Increasing temperature of the inlet coolant temperature produced a side effect in form of increased temperature of the servers and rack. Whenever the coolant inlet temperature is greater than the ambient air temperature, in case of PSNC data center was 25 °C, the system will dissipate a portion of the heat to the data center air. It is also true in case of our system, even though additional thermal insulation installed in the sides of the rack (see Figure 3).

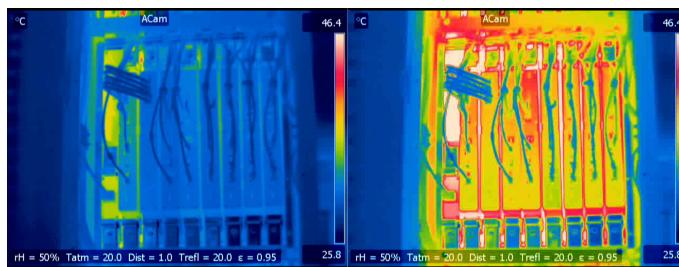


Fig. 3. Temperatures of the cluster with 20 °C inlet (left) and 40 °C inlet (right)

To assess the cooling power of the cooling loop, and to estimate amount of heat dissipated in the Data Center room we used following formula

$$P = V * \rho * c_p * \Delta t_{io} \quad (1)$$

where

- P : power dispersed on the drycooler [W]
- V : flow rate of the coolant [$\frac{dm^3}{min}$]
- ρ : density of the coolant [$\frac{kg}{m^3}$]
- c_p : coolant specific heat [$\frac{J}{kg*K}$]
- Δt_{io} : temperature difference between inlet and outlet temperatures [K]

Using this formula and the data collected by the can calculate a power dispersed on the dry cooler the cooling power of the loop. Looking at the figures (Figure 4 to Figure 8) we can observe how choosing the inlet temperature and delta is influencing the cooling power of the loop in relation to the heat generated by the servers. As our prototype is entirely cooled by the liquid, so there are no fans installed, we assume that 100% of the consumed energy is converted to heat.

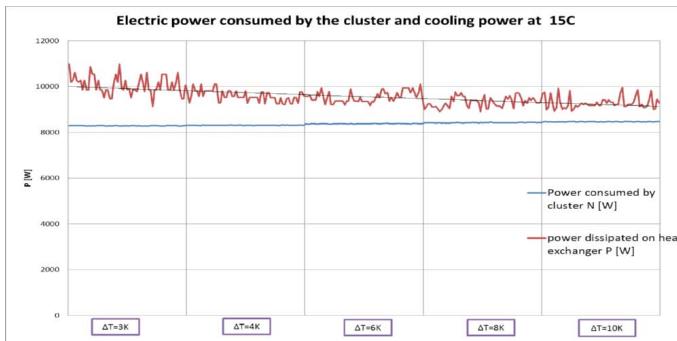


Fig. 4. Cluster power draw and the cooling power of cooling loop at different inlet-outlet deltas with 15 °C inlet coolant

One can see that starting with 30 °C inlet temperature the cooling loop performance delivers less cooling power than the server is generating heat thus making it non-neutral from the server room power budget point of view. With 25 °C and 30 °C the cooling power might be increased by increasing the flow in the secondary loop but it comes at an additional cost. It can also be observed that with the inlet temperatures below the ambient air temperature, the cooling loop delivers more cooling power than the server is producing heat. This means that in such conditions the server is cooling down the data center.

Whenever the difference between set points on inlet and outlet temperatures increases one can observe an increasing erratic behavior of the coolant flow. It is caused by the delay between actions issued by the control system and the reactions in form of change of the temperature of the coolant. Whenever the system detect an increased temperature of the coolant exiting the servers it slowly starts opening the valves to the heat exchanger, then speeds up the

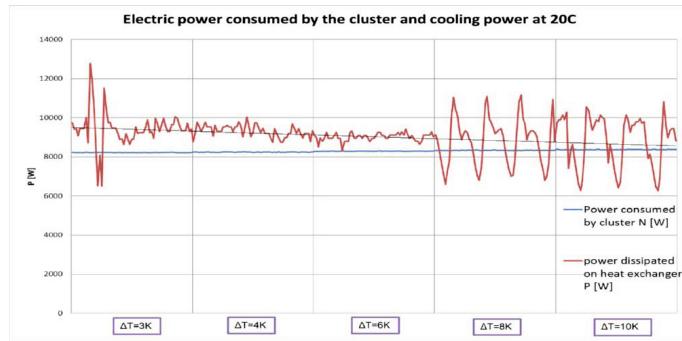


Fig. 5. Cluster power draw and the cooling power of cooling loop at different inlet-outlet deltas with 20 °C inlet coolant

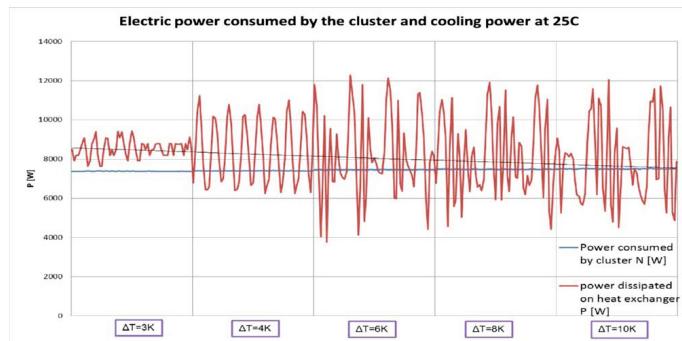


Fig. 6. Cluster power draw and the cooling power of cooling loop at different inlet-outlet deltas with 25 °C inlet coolant

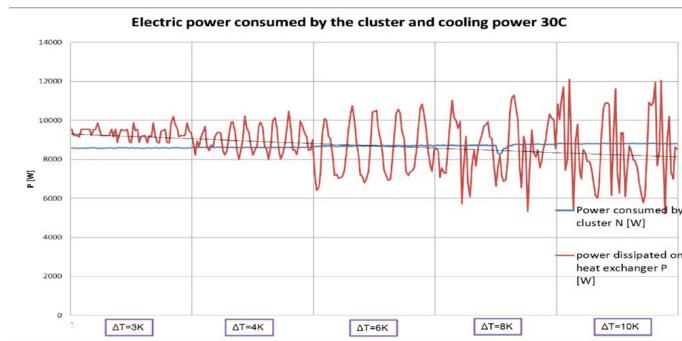


Fig. 7. Cluster power draw and the cooling power of cooling loop at different inlet-outlet deltas with 30 °C inlet coolant

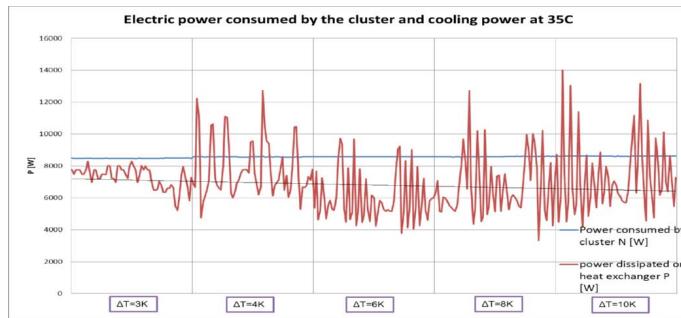


Fig. 8. Cluster power draw and the cooling power of cooling loop at different inlet-outlet deltas with 35 °C inlet coolant

pumps to keep the coolant temperatures within defined range. Because all the sensors are placed in the control rack, the coolant has to traverse more than 50m of pipes before the change of the temperature can be detected. Therefore, when the coolant reaches the sensors it is colder than anticipated so the control system issues a rapid slow-down of the cooling loop and the oscillation occurs. This is affecting the energy efficiency of the cooling loop as the pump settings are higher than necessary. It can be dealt with using an additional coolant tank in the loop that will act as a buffer, unfortunately it was not installed. Figure 8 Coolant flow and cluster inlet and outlet temperatures at 25 °C inlet set point From the overall energy budget of the Data Centers point of view it is crucial to keep the clusters at neutral temperature compared to the ambient air in the room in order not to lose cooling power of the air precision cooling which is less efficient than the dedicated cooling loop. Because we wanted to keep the energy consumption of the dedicated loop also as low as possible we decided to define a preferred inlet-outlet delta for each o of the temperature intervals. For the 15 – 25 °C inlet temperatures the maximum delta of 10K was selected. The border intervals of 25 °C-30 °C inlet were more complicated therefor after

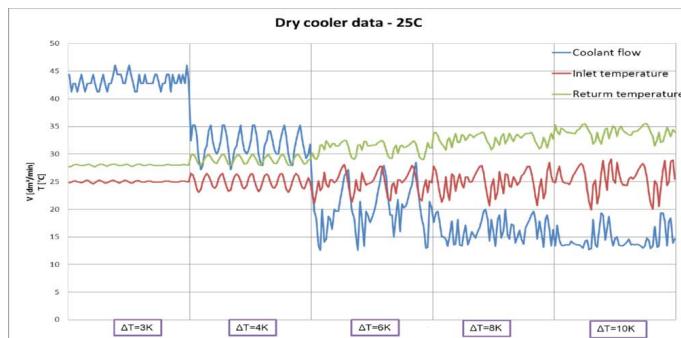


Fig. 9. Coolant flow and cluster inlet and outlet temperatures at 25 °C inlet set point

looking at the data describing flows and temperatures (Figure 9 and Figure 10) and energy consumption of the pump (Figure 2) a delta of 8K for 25 °C and of 6K for 30 °C and above. This allowed us to keep the energy consumed by the cooling loop in worst case below 5% of the total energy consumed by the server-cooling loop pair. During normal server operation, the inlet coolant temperature is set automatically based on the external temperature setting it 1,5K above measured value to account for the dry cooler efficiency.

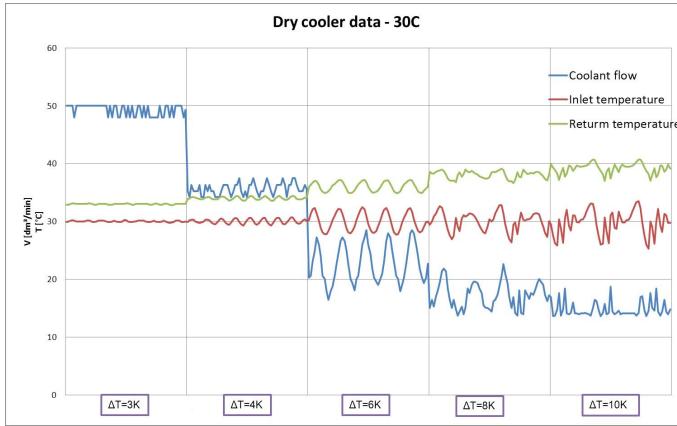


Fig. 10. Coolant flow and cluster inlet and outlet temperatures at 30 °C inlet set point

5 Conclusions and Future Work

The experiment proved that while the general idea of using warm water as a coolant may bring great benefits in terms of energy bill savings, one have to carefully think about what the right coolant temperature is, taking into account the server allowed temperature range and the local climate. Using a constant temperature solution with predefined temperatures requires compromises. The higher temperature one chooses, the bigger portion of the year may be run in free cooling mode (Figure 12) but it affects the efficiency of the cooling loop (Figure 11) and in a slight way also the energy consumed by the servers.

In case of the PSNC prototype, in order to cover all year with a constant temperature solution we would have to go for more than 35 °C inlet temperatures which would bring our system to less efficient coolant ranges. Fortunately, our heat exchanger is equipped with an adiabatic cooling module (data on Figure 11 does not take into account the adiabatic support) and with its support we were able to lower the temperature returning from the dry cooler to 5K below the external temperature. This in turn allowed us to operate with a maximum of 35 °C inlet even during the hottest summer hours. During this time the efficiency of the dedicated cooling loop was around 85%, therefore only 15% of heat was cooled by the facility precision air cooling. Thanks to the innovative approach to

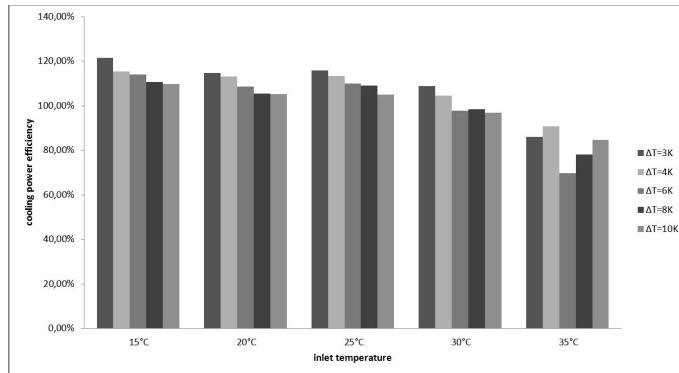


Fig. 11. Cooling power relative to power consumed by the servers

cooling loop management, the system worked in perfect conditions, up to 25 °C inlet temperature, for 90% of hours during the 12 month test period. This shows how beneficial using variable temperature cooling might be. Taking into account that the system had to deal with high (above 35 °C) range of temperatures only for about 60 hours, even with additional cooling power that was used by the facility cooling, the average overhead on the cooling of the prototype cluster was around 2%. Considering that the traditional precision air cooling deployed in our data center is introducing 35-40% of overhead the direct-variable-temperature cooling was approximately 20 times more efficient.

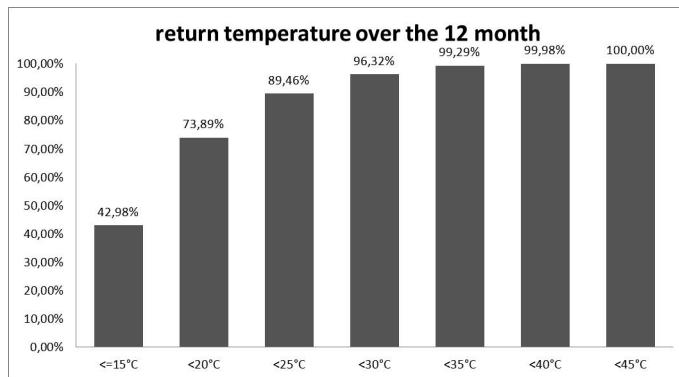


Fig. 12. Distribution of the number of minutes when given return temperature may be kept in a free-cooling mode

This solution might be even more effective if the control system, instead of working on predefined discrete intervals, would optimize the cooling loop behavior basing on efficiency function. This solution might also reduce the coolant

flow oscillation mentioned in the previous chapter. This idea is currently being worked on.

We are also considering a tighter coupling of cooling loop and servers control systems. In order to mitigate problems with high temperatures during the hot summer days, the computing system might reduce its power consumption by reducing the clock of the processors or even suspending/hibernating the nodes for short times. While the second option might be applied only to limited, non-essential nodes, the former seems to be useful in almost all scenarios.

References

1. Poss, M., Nambair, R.O.: Energy Cost, the Key Challenge of Today's Data Centers: A Power Consumption Analysys of TPC-C results
2. [http://www.eweek.com/c/a/IT-Infrastructure/
How-to-Optimize-the-Energy-Efficiency-of-Your-Server/2/](http://www.eweek.com/c/a/IT-Infrastructure/How-to-Optimize-the-Energy-Efficiency-of-Your-Server/2/)
3. http://hightech.lbl.gov/documents/data_centers/airflow-doe-femp.pdf
4. <http://www.google.com/about/datacenters/efficiency/external/>
5. Januszewski, R., Gilly, L., Yilmaz, E., Auweter, A., Svensson, G.: Cooling making efficient choices
6. Peaty, D., Davidson, T.: Data Centers Datacom Airflow Patterns ASHRAE Journal (April 2005)
7. <http://www.42u.com/cooling/data-center-temperature.htm>
8. Partnership for Advanced Computing in Europe, <http://prace-project.eu/>
9. <http://www.iceotope.com/>

A Case Study of Energy Aware Scheduling on SuperMUC

Axel Auweter¹, Arndt Bode¹, Matthias Brehm¹, Luigi Brochard²,
Nicolay Hammer¹, Herbert Huber¹, Raj Panda³,
Francois Thomas², and Torsten Wilde¹

¹ Leibniz Supercomputing Centre of the Bavarian Academy of Sciences
and Humanities, Garching, Germany

axel.auweter@lrz.de

² IBM System and Technology Group, Montpellier, France

³ IBM System and Technology Group, Austin, Texas, USA

Abstract. In this paper, we analyze the functionalities for energy aware scheduling of the IBM LoadLeveler resource management system on SuperMUC, one of the world’s fastest HPC systems. We explain how LoadLeveler predicts execution times and the average power consumption of the system’s workloads at varying CPU frequencies and compare the prediction to real measurements conducted on various benchmarks. Since the prediction model proves to be accurate for our application workloads, we can analyze the LoadLeveler predictions for a large fraction of the SuperMUC application portfolio. This enables us to define a policy for energy aware scheduling on SuperMUC, which selects the CPU frequencies considering the applications’ power and performance characteristics thereby providing an optimized tradeoff between energy savings and execution time.

Keywords: energy aware scheduling, power modelling, resource management, HPC.

1 Motivation

Recent trends in High-Performance Computing (HPC) reveal that energy efficiency of supercomputers is a challenge more than ever. While performance growth in computing is typically expressed by Moore’s law, the world’s most powerful HPC systems have been outperforming Moore’s law for years. Figure 1 illustrates this trend together with its effect on power consumption. While the compute power of the top 10 systems has increased on average by a factor of 1.9 every year during the last 6 calendar years, the power consumption of the systems has increased as well on average by a factor of 1.2 every year. The power consumption of the leading edge supercomputers has reached a level of more than 10 MegaWatts (MW), yet it continues to grow.

Due to rising energy prices, for reasons of climate protection, and due to the involved technical challenges and limitations it is commonly accepted that the

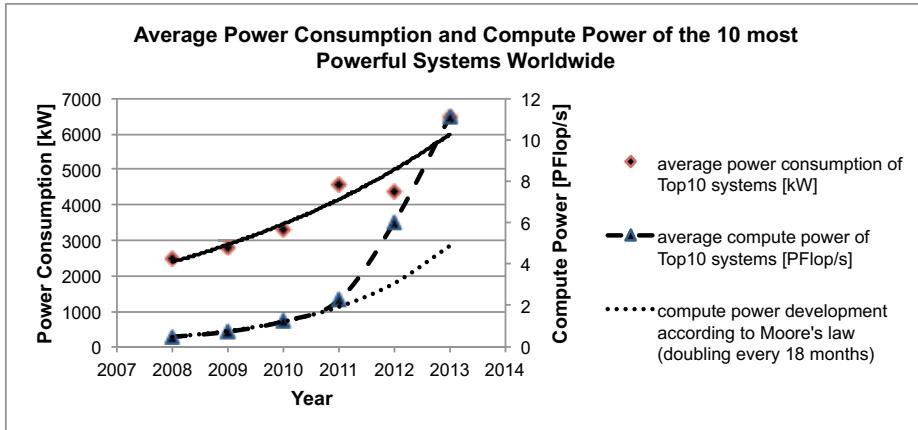


Fig. 1. Average power consumption and compute power of the 10 most powerful HPC systems worldwide according to TOP500 [1]

power consumption of sustainable many-Peta to Exascale computing needs to stay in a 1 to 20 MW range. This definite power challenge was first expressed in “Exascale Computing Study: Technology Challenges in Achieving Exascale Systems” [2], which became a main driver for energy efficiency research in the field of HPC. Today, increasing the energy efficiency of HPC systems is one of the main goals of the HPC community.

In our previous work, we have described the 4-pillar framework for energy efficient HPC [3]. It consists of the pillars “Energy Efficient Building Infrastructures”, “Energy Efficient HPC System Hardware”, “Energy Aware System Software”, and “Energy Efficient HPC Applications”. While the framework proposes specific optimizations within each pillar, it particularly encourages optimizations crossing the various pillars. This paper presents our approach of reducing the energy consumption of supercomputers with the use of energy aware resource management software that relies on thorough power monitoring hardware and detailed application performance monitoring, thus being a cross-pillar optimization spanning two of the four pillars: HPC system hardware and energy aware system software.

Our approach builds on the ability to vary processor clock frequencies and supply voltages according to the actual demand. This technology, commonly referred to as Dynamic Voltage and Frequency Scaling (DVFS), has been proposed by microprocessor designers years ago and is nowadays implemented in every modern CPU [4] [5]. Thus, DVFS has also been the subject of many publications in the field of energy efficient computing.

We describe the model used by the batch scheduling software IBM Load-Leveler [6] to predict the power and performance of applications running at different processor frequencies on a cluster of standard x86 servers [7]. While the prediction model itself could be integrated into any modern workload

management software, LoadLeveler is currently the only tool to provide this feature for production-level environments. We also present how we apply the prediction model to reduce the energy consumption of SuperMUC [8] operated by Leibniz Supercomputing Centre of the Bavarian Academy of Sciences and Humanities (LRZ) in Garching, Germany. SuperMUC runs on behalf of the Gauss Centre for Supercomputing (GCS) and is available to European scientists through the Partnership for Advanced Computing in Europe (PRACE). It currently consists of 9,421 nodes connected via FDR10 InfiniBand and provides more than 155,000 Intel Xeon processor cores as well as 340 TByte of main memory to its users with an extension to be installed by the end of 2014. With a Linpack performance of 2.9 PetaFlops ($=2.9 \times 10^{15}$ Floating Point Operations per Second), SuperMUC is still one of the world's fastest HPC systems despite not relying on special compute accelerators.

This study is performed on SuperMUC's thin nodes. Each thin node is diskless and features two Xeon E5-2680 CPUs with 8 cores each running at up to 2.7 GHz and 32GB of main memory configured in 8 DDR3 DIMMs running at 1600MHz. In addition to the Infiniband network, the thin nodes are connected via Gigabit Ethernet for booting and remote management.

2 LoadLeveler Prediction Model

SuperMUC provides a thorough power monitoring infrastructure with various measurement points in the power distribution chain. Out of all components contributing to SuperMUC's total power consumption, the compute nodes prevail the other subsystems with a power draw of up to 2.7 MW (the I/O and networking subsystems exhibit a constant power consumption of only 116 kW and 74 kW respectively). Therefore, we concentrate on modelling and optimizing the power consumed by SuperMUC's compute nodes consisting of standard x86 server hardware. Where measuring node power is necessary, we use SuperMUC's paddle cards. The paddle cards measure the consumed energy of each node at the power supply unit. By providing energy values, measuring with paddle cards causes only little interference with the applications running on the system since the energy counter needs only be sampled when applications start and end. The mean power consumption can then be derived from dividing the runtime into the consumed energy.

To predict the power and performance of applications running at different frequencies, our model takes into account the compute nodes and the application. For simplicity, we analyze application behavior as a whole, focusing on the mean power and performance during the runtime. Looking first at power consumption, we know that the power consumed by an application when running on a server varies with the node configuration and the type of application. In earlier experiments [9], we found that node power consumption consists of two parts: a dynamic part which relates to the activity of the processor, cache, memory, etc. and a static part which is unrelated to any activity and originates mostly from the south bridge chip. This static part is dominating the power consumption

when the node is idle. We also found that the processor power consumption varied according to the average number of Cycles Per Instruction (*CPI*) required during the application run and the memory power consumption varied according to the total read and write memory bandwidth (*GBS*) of the application. This is equivalent to the commonly accepted characterization of application behavior under frequency scaling depending on memory or compute boundedness.

Based on these findings, the power consumption of an application on a particular compute node at frequency f_n can be predicted from a measurement performed at a reference frequency f_0 :

$$PWR(f_n) = A_n * GIPS(f_0) + B_n * GBS(f_0) + C_n \quad (1)$$

where:

- $PWR(f_n)$ is the node power at frequency f_n
- $GIPS(f_0)$ and $GBS(f_0)$ are the instruction rate (number of Giga Instructions Per Second with $GIPS = Frequency/CPI$) and the memory bandwidth (GigaBytes per Second) measured when the application is run at reference frequency f_0
- A_n , B_n and C_n are the power coefficients characterizing a given platform at frequency f_n with:
 - A_n representing the power consumed for processing instructions
 - B_n representing the power consumed for transferring data
 - C_n representing the static power consumption

In this model, coefficients A_n , B_n , and C_n are system-specific, while *GIPS* and *GBS* are application-specific and need to be measured once for each application at its first run at the reference frequency f_0 .

To calculate the coefficients A_n , B_n , and C_n , LoadLeveler uses an initialization phase (which is run once for a given reference frequency and a given system hardware configuration). In this phase, LoadLeveler runs a set of compute kernels which provide a large spectrum of *CPI* and *GBS* characteristics. For each compute kernel, LoadLeveler measures the power consumption of the node at all possible CPU frequencies. Then, with a multiple linear regression analysis using least squares fitting, the values of A_n , B_n and C_n are approximated such that the node power measured satisfies Equation 1 over all kernels at a given frequency.

A similar method is used to predict the runtime of the application at any frequency from the application runtime at frequency f_0 and hardware coefficients D_n , E_n , and F_n which are calculated in the same initialization phase.

We first derive *CPI*(f_n) based on *CPI*(f_0):

$$CPI(f_n) = F_n + D_n * CPI(f_0) + E_n * TPI(f_0) \quad (2)$$

where:

- $CPI(f_0)$ is the measured CPI at frequency f_0
- $TPI(f_0)$ is the number of memory Transactions Per Instruction at frequency f_0 with:
 $TPI = \text{Membytes}/\text{Instructions}/\text{Cache_Line_Size}$ where:
 - Membytes is the number of bytes written and read from memory by the application
 - Instructions the number of instructions executed by the application
 - Cache_Line_Size the cache line size in bytes
- F_n , D_n and E_n are the coefficients resulting from the least squares fitting

We then compute the application runtime at frequency f_n by:

$$TIME(f_n) = TIME(f_0) * CPI(f_n)/CPI(f_0) * (f_0/f_n) \quad (3)$$

3 LoadLeveler Energy Aware Scheduling Implementation

The energy aware scheduling features in LoadLeveler have been implemented using a three phases approach. The initialization phase has already been described in the previous section. At its end, LoadLeveler stores the coefficients A_n , B_n , C_n , D_n , E_n , and F_n for all nodes and all possible frequencies in a database.

For phases two and three, LoadLeveler needs to be able to identify jobs. In this context, the combination of an executable and a set of input parameters or input files is considered as a job. The power-performance characteristics of a job can depend on both, the executable and the input parameters, and LoadLeveler cannot determine whether two jobs will feature a similar behavior when varying the CPU frequency. Therefore, we request the user to provide a tag which will uniquely identify groups of similar jobs. Jobs can be considered similar, for example, if minor physical parameters of the simulation are changed but the algorithm of the application remains the same. In contrast, different tags should be assigned when solving a completely different type of problem. We refer to these user-defined tags in LoadLeveler as “Energy Tags”.

The second phase takes place when a job with an unknown energy tag is run and forces the job to run at the reference frequency. During this run, LoadLeveler monitors the application and calculates CPI , GBS , and $GIPS$. It also records runtime, power consumption over time, and total energy consumption of the job. All information is stored in LoadLeveler’s database under the user-assigned energy tag.

The third phase starts when a similar job (i.e. with the same energy tag) is submitted. By using the prediction model and the application-specific parameters obtained in the second phase, LoadLeveler selects an appropriate CPU frequency for the application execution according to a configurable policy.

4 Validation of the LoadLeveler Prediction Model

To validate the prediction model of LoadLeveler, we compare the LoadLeveler predictions for runtime and power consumption at varying CPU frequencies

with real-world measurements. For this test, we use four scientific applications: Quantum ESPRESSO [10], Gadget [11], Seissol [12], and WaLBerla [13]. The applications have been chosen from the LRZ benchmark suite, a collection of applications that represents well the typical workloads of LRZ's users. In addition to these applications, two synthetic benchmarks, PMATMUL and STREAM are used to represent the two corner cases of purely CPU and purely memory bound applications.

Like LoadLeveler, we rely on SuperMUC's paddle cards for the real-world power measurements. To ensure the accuracy of the measurements, we have verified the numbers from the paddle cards using higher-level instrumentation points integrated in SuperMUC's power distribution units (PDUs). Repetitions of our experiments yield a mean square error of the power measurements below 0.5%.

4.1 Quantum ESPRESSO

Quantum ESPRESSO (opEn Source Package for Research in Electronic Structure, Simulation, and Optimization) is a tool suite for electronic-structure calculation and material modelling at the nanoscale. For our benchmark, we use a small input data set for Quantum ESPRESSO's plane-wave self-consistent field (PWscf) tool using a hybrid MPI+OpenMP parallelization scheme on 16 nodes with 4 MPI ranks per node and 4 OpenMP threads per task.

The graphs in Figure 2 show that LoadLeveler's prediction model can estimate the application runtime and power consumption for different CPU frequencies accurately. On average, the prediction differs from the actual measurement by less than 3% for the application runtime and by less than 1% for the application power consumption.

4.2 Gadget

Gadget is a numerical N-body-Magneto-Smoothed-Particle-Hydrodynamics code widely used for cosmological structure formation simulations. For our tests, we use the latest hybrid MPI+OpenMP version of Gadget-3 running a pure dark matter (N-body) simulation on 8 compute nodes, with 4 MPI ranks per node and 4 OpenMP threads per MPI task.

In the Gadget test case, the prediction differs from the actual measurement in all cases by less than 1% for the runtime (0.39% on average) and by less than 3% for the power consumption (1.11% on average).

4.3 Seissol

SeisSol is a numerical application simulating seismic wave phenomena. For the presented measurements, we run an elastic wave propagation layer over half-space test-case using a hybrid MPI+OpenMP parallelized version of SeisSol [14]. The code runs on 16 computes nodes with 1 MPI rank per node and 16 OpenMP threads per MPI task.

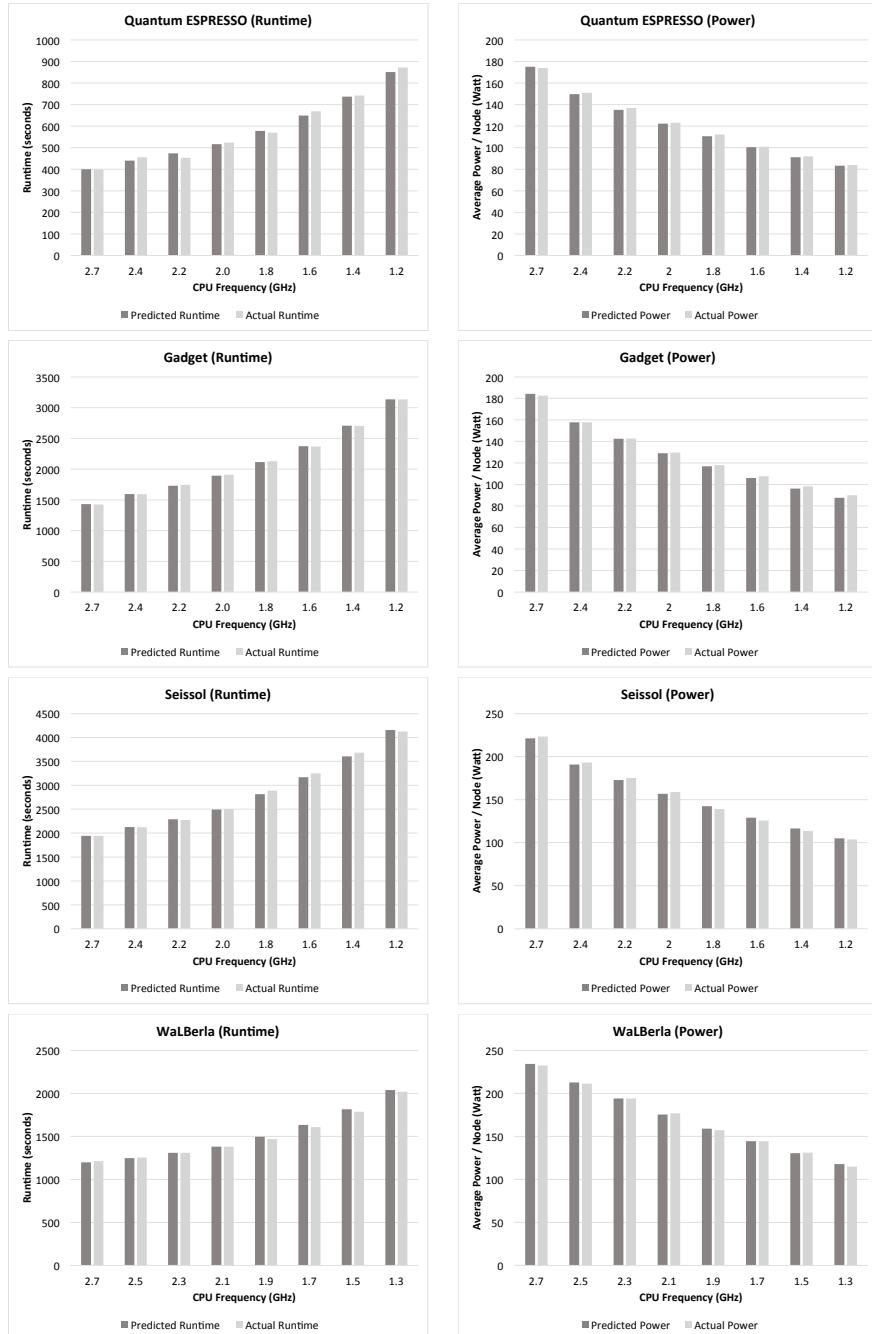


Fig. 2. Comparison of LoadLeveler predictions for application runtime and node power consumption to actual measurements under selected application benchmarks on SuperMUC

In the Seissol test case, the prediction differs from the actual measurement in all cases by less than 3% for the runtime (1.16% on average) and by less than 3% for the power consumption (1.66% on average).

4.4 WaLBerla

WaLBerla is a simulation code for various physics applications implementing a Lattice-Boltzmann solver. The test case is a pure MPI application with 1024 MPI ranks on 64 nodes.

In the WaLBerla test case, the prediction differs from the actual measurement in all cases by less than 2% for the runtime (0.95% on average) and by less than 3% for the power consumption (0.81% on average).

4.5 PMATMUL

PMATMUL is a simple parallel dense matrix multiplication benchmark. It runs as a pure MPI application with 1024 MPI ranks on 64 nodes. Its implementation uses a block-wise approach and performs the matrix multiplication using Intel's MKL library in the inner loop.

The graphs in Figure 3 show that the power predictions of LoadLeveler are accurate on PMATMUL (less than 1% deviation from the prediction), yet runtime predictions deviated by up to 7%. It can be observed that runtime predictions get worse at low CPU frequencies.

4.6 STREAM

STREAM is a well-known memory bandwidth benchmark that was taken from the LikwidBench tool of the Likwid performance toolsuite [15]. We run it on a single node with 16 OpenMP threads.

STREAM shows the least accurate prediction with up to 7% deviation for the predicted runtime and up to 5% deviation for the predicted power consumption. In contrast to PMATMUL, however, predictions are getting worse at high CPU frequencies.

4.7 Summary of Benchmark Results

In our six test cases, we have shown that the LoadLeveler prediction model is accurate for full scientific applications. The slightly weaker accuracy of the prediction during the synthetic benchmarks can be explained by the fact that the two selected benchmarks represent corner cases that do not exhibit a reasonable mix of compute and memory utilization for which the LoadLeveler prediction model was designed. In addition, a potential error source for synthetic benchmarks is an increased interference by the instrumentation code during energy tag creation when CPU performance or memory bandwidth are fully saturated. Whether a refined prediction model could yield even more accurate results will be subject to further research.

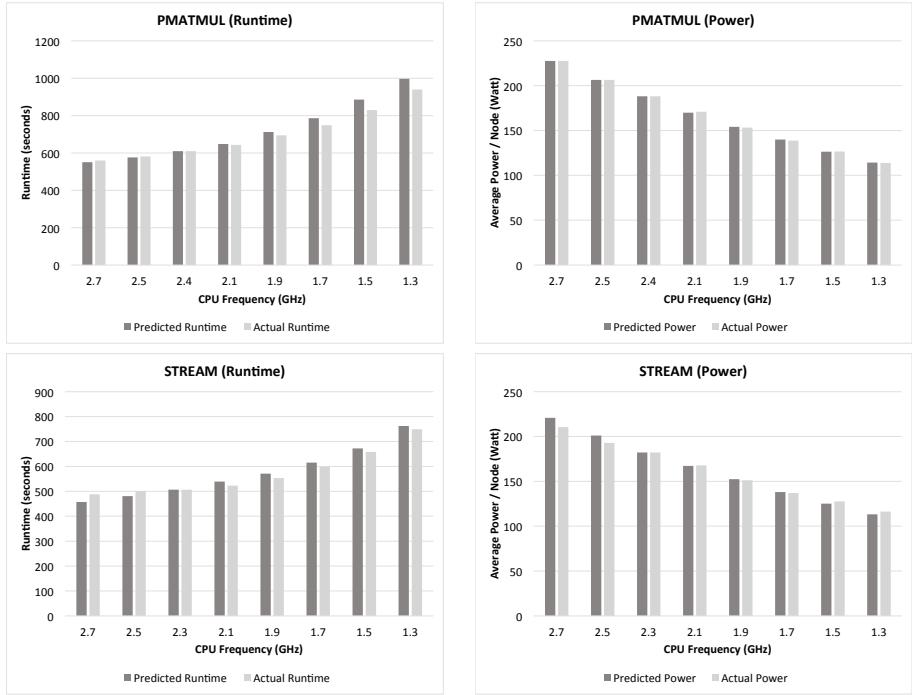


Fig. 3. Comparison of LoadLeveler predictions for application runtime and node power consumption to actual measurements under the two synthetic benchmarks PMATMUL and STREAM on SuperMUC

5 Towards a Reasonable Energy Savings Policy at LRZ

The experiments described above show that LoadLeveler reliably predicts runtime and power consumption of common scientific applications on SuperMUC at varying CPU frequencies. Hence, the data of LoadLeveler can also be used to characterize the power-performance behavior of all applications on SuperMUC.

The goal of such an analysis is twofold: first, we would like to know which CPU clock frequency can be considered as a reasonable default setting on our system. This frequency should be chosen in order to provide a good power-performance tradeoff for most applications and can be applied to applications for which no energy tag has been assigned yet. Second, we would like to define a policy that allows certain jobs to deviate from that default frequency setting if their power-performance tradeoff has its optimum at a given frequency other than the default. Note that a definition of “optimum” is intentionally skipped as it should include the policy decision of a computing center to trade in computational throughput for energy savings.

5.1 Analyzing the LRZ Application Portfolio

To gather the necessary data, SuperMUC users were encouraged to assign LoadLeveler energy tags to each application. Thanks to our users, a total of 469 energy tags were generated. For sanitizing the data, we removed from our dataset all energy tags that were generated for applications which ran only 10 minutes or less. This way, we hope to remove most runs consisting of small benchmarks or failed jobs. We then went on to remove fourteen obvious outliers which we could trace back to malfunctions in the power measurement hardware at the time of the energy tag creation. After sanitization, 236 energy tags remained for further analysis.

Since LoadLeveler predicts absolute values for the application runtime and average power consumption, we normalize all predictions relative to 2.7 GHz. Figure 4 shows the predicted increase in application runtime of all applications in the dataset when downscaling the CPU frequency from 2.7 GHz to 1.2 GHz. The runtime increase of each application is shown as a gray line with overlapping lines resulting in a gradual darkening of the color. The average over all application runtime predictions is shown as white dashes.

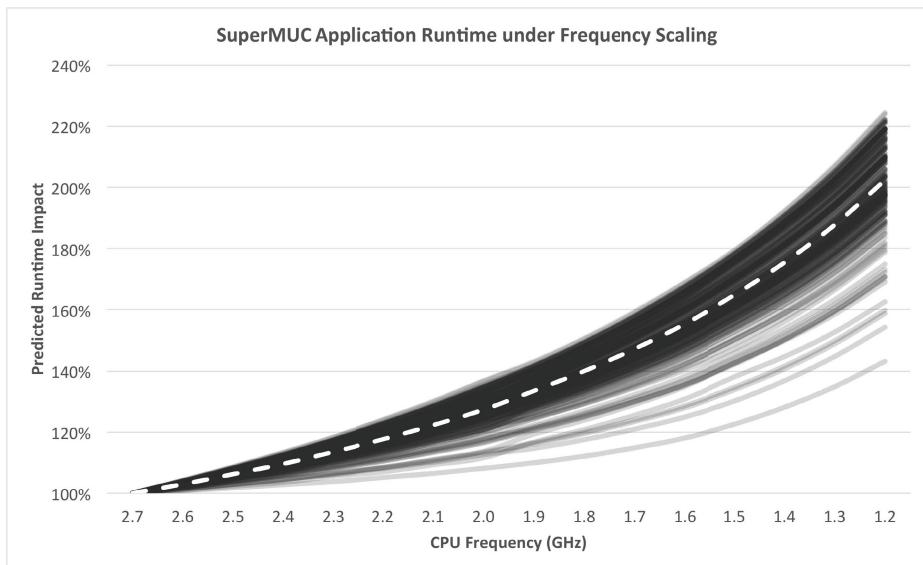


Fig. 4. SuperMUC application runtime under CPU frequency scaling

Figure 5 shows the predicted decrease in power consumption when downscaling the CPU frequency. The visualization approach is similar to Figure 4 with each application being represented by a gray line resulting in a gradual darkening of the graph in case of overlapping lines. Again, the average of all applications is drawn as white dashes.

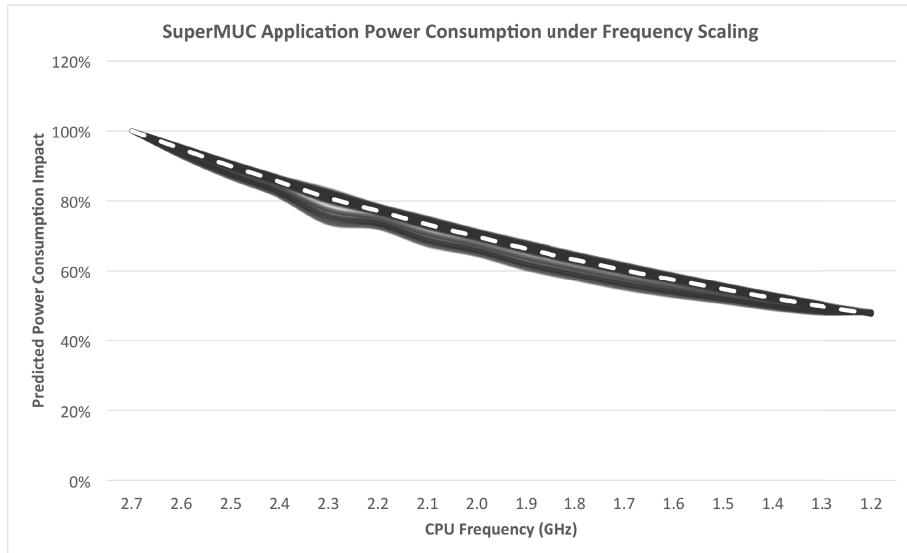


Fig. 5. SuperMUC application power consumption under CPU frequency scaling

From the two above figures we can conclude that the power consumption behavior over the frequency range varies only a little between the applications. Instead it is merely determined by the selected frequency. On the other hand, the runtime behavior seems largely dependent on the application: at the lowest frequency of 1.2 GHz, the performance impact compared to 2.7 GHz can be as little as 40% in the best case and as much as 120% in the worst case. On average, the runtime of applications on SuperMUC doubles when scaling from 2.7 GHz to 1.2 GHz while the power consumption is halved.

5.2 Finding an Optimal Default Frequency

One observation from Figure 4 and Figure 5 is that the increase in runtime and the decrease in power consumption are non-linear over the CPU frequency range. This results in a theoretical optimum when analyzing the Energy-to-Solution (*ETS*) of applications defined as:

$$ETS = (\text{Average Power per Node}) * \#Nodes * \text{Runtime} \quad (4)$$

Energy-to-Solution is a metric to consider when looking for a good default frequency for SuperMUC. Figure 6 shows the Energy-to-Solution that results from averaging over our dataset of runtimes and power consumptions as predicted by LoadLeveler's energy tags. It suggests that SuperMUC applications will consume the least energy when running at a frequency of 1.8 GHz.

LoadLeveler implements a “Minimize Energy-to-Solution” policy where each application is run at the CPU frequency for which the prediction yields the lowest

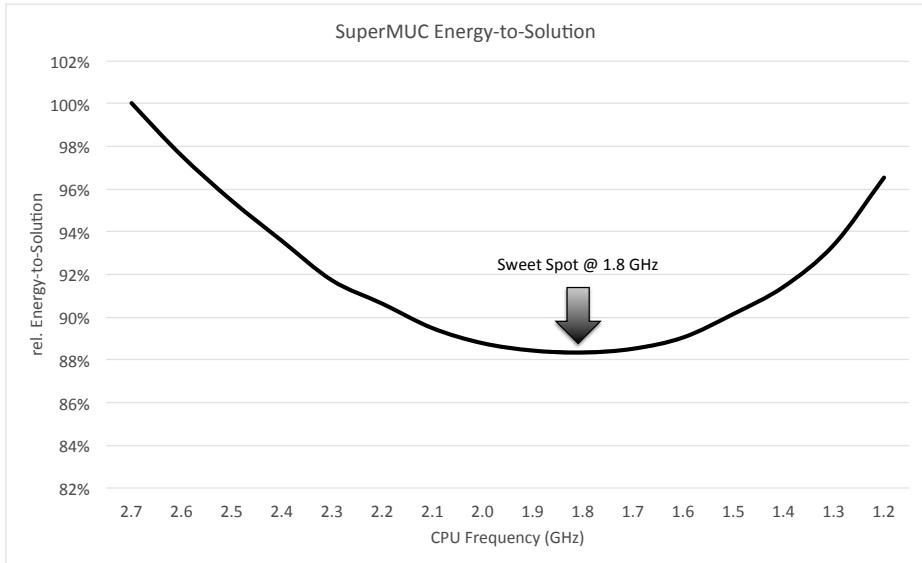


Fig. 6. Predicted Energy-to-Solution averaged across all SuperMUC applications

ETS. It sounds like a reasonable choice to minimize Energy-to-Solution (with the effect of reducing Energy-to-Solution by almost 12% on average). However, defaulting to 1.8 GHz would neglect that the average runtime increase at 1.8 GHz is almost 40%. This means that while the individual result can be obtained with significantly lower energy costs, the productivity of the system is reduced drastically. Using the “Minimize Energy-to-Solution” policy of LoadLeveler is therefore not desirable for a production environment like LRZ’s.

Many researchers suggest using the Energy Delay Product (*EDP*) to account for the loss in productivity when trying to optimize energy consumption ([16], [17]). The Energy Delay Product puts a stronger weight on performance by being defined as:

$$EDP = (\text{Average Power per Node}) * \# \text{Nodes} * \text{Runtime}^2 \quad (5)$$

Figure 7 shows the Energy Delay Product obtained in a similar way to Figure 6. The optimal *EDP* is at 2.7 GHz. However, weighting runtime in this way is arbitrary and overvalues the drawbacks of reduced application performance. Finally, no technical optimum can be defined, leaving it as a policy decision to the data center to find a reasonable tradeoff between computational throughput and optimized Energy-to-Solution.

For LRZ, both Energy-to-Solution and Energy Delay Product have been considered. Figure 6 shows that *ETS* decreases sharply from 2.7 GHz to 2.2 GHz while Figure 7 shows that the curve for *EDP* is flat between 2.7 GHz and 2.5 GHz. Thus, 2.5 GHz is a good candidate for our default frequency. However, we selected the more “aggressive” default frequency of 2.3 GHz on SuperMUC

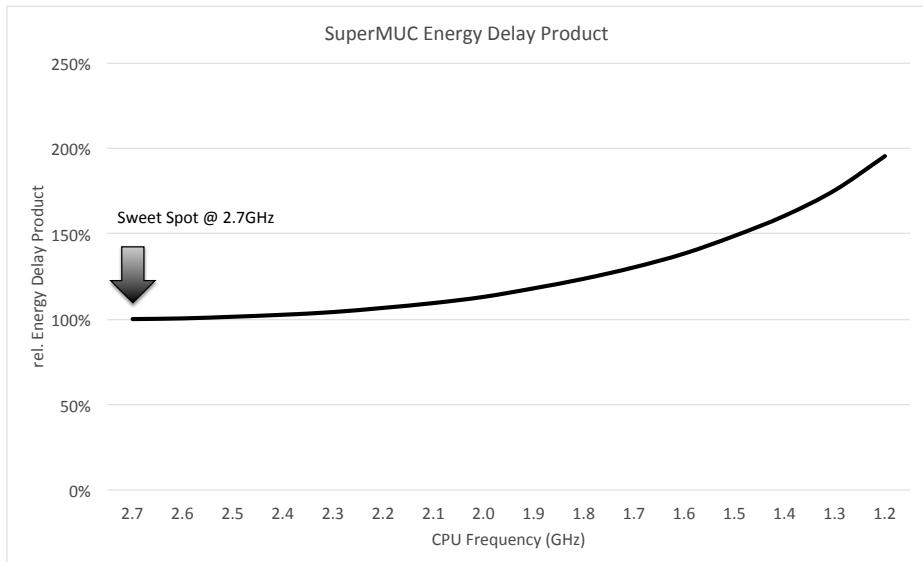


Fig. 7. Predicted Energy Delay Product averaged across all SuperMUC applications

because we would allow certain applications to run at full speed as will be explained in the next section.

If our policy was to use the 2.3 GHz default setting for all applications on SuperMUC, the average application would save more than 19% of power with a performance impact of 13%, resulting in energy savings of over 8% compared to running at 2.7 GHz. Being selected as a reasonable default setting, LoadLeveler also assigns the default frequency of 2.3 GHz to all jobs that do not explicitly specify an energy tag.

5.3 How to Run at Full Speed

While reducing the default frequency to 2.3 GHz for improved energy efficiency is well justified, a small group of heavily optimized applications would show significant performance gains when running at higher frequencies. In contrast to the majority of the applications in the LRZ application portfolio, these applications would suffer from an over-average performance penalty in case their execution is slowed down by lowering CPU frequencies.

Therefore, LoadLeveler features another policy called “Minimize Time-to-Solution”. This enables application developers to benefit from higher-than-default frequencies if their application’s energy tag predicts that certain performance thresholds can be met at higher frequencies. At LRZ, the thresholds are configured as:

- 5% runtime decrease at 2.5 GHz (97% of the performance gain that an ideal application would experience when going from 2.3 GHz to 2.5 GHz)

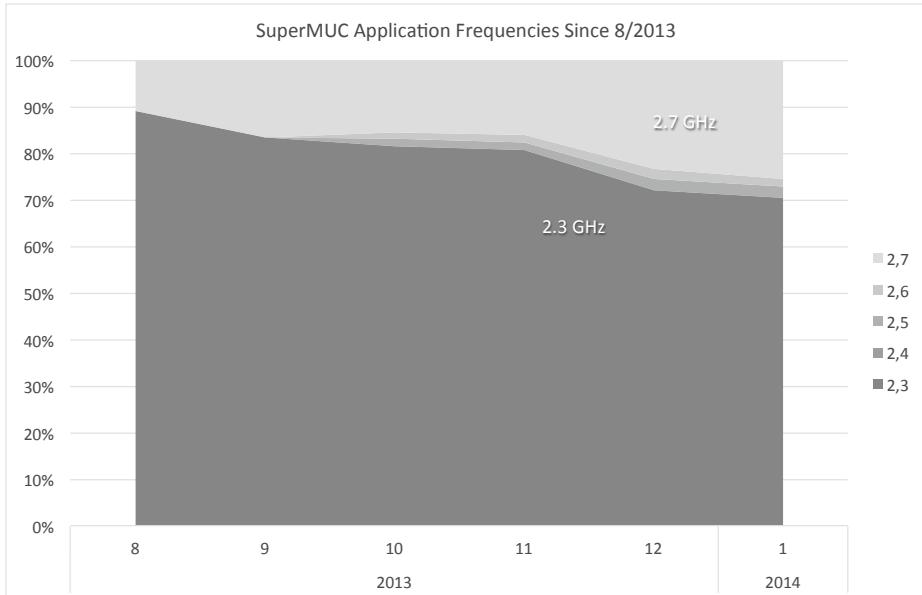


Fig. 8. CPU frequencies at which SuperMUC applications are run since August 2013

- 12% runtime decrease at 2.7 GHz (96% of the performance gain that an ideal application would experience when going from 2.3 GHz to 2.7 GHz)

In the case where LoadLeveler’s prediction estimates at least a 5% shorter runtime at 2.5 GHz over the 2.3 GHz default, the application will run at 2.5 Ghz. And, if the even higher threshold of a 12% shorter runtime at 2.7 Ghz is predicted, the application will run at 2.7 Ghz.

Using this approach, we ensure that our default policy of 2.3 GHz does not impose penalties on applications that benefit from higher CPU frequencies. In addition, we create an incentive for our users to further optimize their scientific codes running on SuperMUC. Figure 8 shows the CPU frequency distribution of all jobs running on SuperMUC since August 2013 when the new policy was introduced. It shows that since its introduction, more and more jobs benefit from the ability to run at full speed.

However, as of January 2014, with 70% of the applications running at 2.3 GHz, our policy still helps us have a 6% energy saving than when all applications ran at full speed.

6 Conclusion

In this paper, we have explained the implementation of energy aware scheduling on SuperMUC using IBM LoadLeveler. We have shown that LoadLeveler

successfully predicts application runtimes and node power consumption for different frequencies allowing it to run each application at the CPU frequency that is considered to be optimal according to LRZ's policies.

We have analyzed the power-performance profiles of a representative amount of applications running on SuperMUC in production to justify our selection of policies and thresholds. Finally, we proved that our policies help optimizing the energy efficiency of SuperMUC without penalizing well-optimized and highly scalable codes. While the achievement of 6% energy savings does not seem significant, for LRZ this results in over EUR 200,000 annual savings on electricity costs.

Future work will include evaluating the prediction model on newer hardware (e.g. Intel Haswell CPUs) and refining the approach by using input data of higher resolution for the prediction model (e.g. by using Intel RAPL counters and by selecting the best frequency for each application phase instead of choosing a fixed frequency for the entire application runtime).

Acknowledgments. LRZ's research on energy efficiency in HPC is partly funded by the European Commission under the 7th Framework Programme (FP7/2007-2013) via the projects (grant agreement no.) AutoTune (ICT-248481), DEEP (ICT-287530), Mont-Blanc (ICT-288777), and PRACE (RI-261557, RI-283493).

References

1. TOP500 Supercomputer Sites, <http://www.top500.org/>
2. Bergman, K., Borkar, S., Campbell, D., Carlson, W., Dally, W., Denneau, M., Franzon, P., Harrod, W., Hill, K., Hiller, J., et al.: Exascale computing study: Technology challenges in achieving exascale systems. In: Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO). Tech. Rep. (2008)
3. Wilde, T., Auweter, A., Shoukourian, H.: The 4 pillar framework for energy efficient HPC data centers. Computer Science-Research and Development, 1–11 (2013)
4. Burd, T.D., Brodersen, R.W.: Energy efficient CMOS microprocessor design. In: Proceedings of the Twenty-Eighth Hawaii International Conference on System Sciences, vol. 1, pp. 288–297. IEEE (1995)
5. Horowitz, M., Indermaur, T., Gonzalez, R.: Low-power digital design. In: IEEE Symposium on Low Power Electronics, Digest of Technical Papers, pp. 8–11. IEEE (1994)
6. IBM: Tivoli workload scheduler LoadLeveler,
<http://www.ibm.com/systems/software/loadleveler/>
7. Bell, R., Brochard, L., DeSota, D., Panda, R., Thomas, F.: Energy-aware job scheduling for cluster environments. US Patent 8,612,984 (December 17, 2013)
8. Leibniz Supercomputing Centre of the Bavarian Academy of Sciences and Humanities: SuperMUC supercomputer,
<http://www.lrz.de/services/compute/supermuc/>
9. Brochard, L., Panda, R., Desota, D., Thomas, F., Bell Jr., R.: Optimizing performance and energy of high performance computing applications. Parallel Computing: From Multicores and GPU's to Petascale 19, 455 (2010)

10. Giannozzi, P., Baroni, S., Bonini, N., Calandra, M., Car, R., Cavazzoni, C., Ceresoli, D., Chiarotti, G.L., Cococcioni, M., Dabo, I., et al.: QUANTUM ESPRESSO: a modular and open-source software project for quantum simulations of materials. *Journal of Physics: Condensed Matter* 21(39), 395502 (2009)
11. Springel, V.: The cosmological simulation code GADGET-2. *Monthly Notices of the Royal Astronomical Society* 364(4), 1105–1134 (2005)
12. Käser, M., de al Puente, J., Castro, C., Hermann, V., Dumbser, M.: Seismic wave field modelling using high performance computing (2008)
13. Feichtinger, C., Donath, S., Köstler, H., Götz, J., Rüde, U.: WaLBerla: HPC software design for computational engineering simulations. *Journal of Computational Science* 2(2), 105–112 (2011)
14. Heinecke, A., Breuer, A., Gabriel, A., Pelties, C., Rettenberger, S., Bader, M., Wenk, S.: Optimized kernels for large scale earthquake simulations with SeisSol, an unstructured ADER-DG code. In: *Supercomputing 2013, The International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE (2013)
15. Treibig, J., Hager, G., Wellein, G.: Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In: *2010 39th International Conference on Parallel Processing Workshops (ICPPW)*, pp. 207–216. IEEE (2010)
16. 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*, p. 34. IEEE Computer Society (2005)
17. Freeh, V.W., Lowenthal, D.K., Pan, F., Kappiah, N., Springer, R., Rountree, B.L., Femal, M.E.: Analyzing the energy-time trade-off in high-performance computing applications. *IEEE Transactions on Parallel and Distributed Systems* 18(6), 835–848 (2007)

Exploiting SIMD and Thread-Level Parallelism in Multiblock CFD

Ioan Hadade and Luca di Mare

Whole Engine Modelling Group

Rolls-Royce Vibration UTC

Imperial College London

South Kensington, SW7 2AZ, London, United Kingdom

{i.hadade,l.di.mare}@imperial.ac.uk

<http://www3.imperial.ac.uk/vutc>

Abstract. This paper presents the on-node performance tuning of a multi-block Euler solver for turbomachinery computations.

Our work focuses on vertical and horizontal scaling within an x86 multi-socket compute node by exploiting the fine grained parallelism available through SIMD instructions at core level and thread-level parallelism across the die through shared memory. We report on the challenges encountered in enabling efficient vectorization using both compiler directives and intrinsics with an emphasis on data structure transformations and their performance impact on vector computations.

Finally, we present the solver performance on different grid sizes running on Intel Sandy Bridge and Ivy Bridge processors.

Keywords: computational fluid dynamics, performance analysis, SIMD, AVX, OpenMP, vectorization, high performance computing, parallel programming.

1 Introduction

Historically, improvements in the speed of fluid dynamics simulations have been achieved by advancements in both the applied numerical methods and underlying hardware. The latter has been a greater contributor than the former due to the constant increase in attainable performance with every new generation of processors. Recently, growing importance has been placed on in-core optimisations such as Single-Instruction-Multiple-Data (SIMD) execution [1],[2],[3]. Although older microprocessors have been able to perform SIMD operations for over fifteen years through a number of instruction set extensions such as MMX or Streaming SIMD Extensions (SSE)[4], the number of vector registers and their width has remained fairly constant for over a decade. Furthermore, the majority of vector instructions found in these extensions were aimed at single precision floating point data and operations which made their usability limited for some scientific applications. Intel's Sandy Bridge micro architecture doubles the vector register

width to 256-bits from the previous 128-bits and introduces a new SIMD instruction set extension in the form of Advanced Vector eXtensions (AVX)[5]. The new extension offers an increase in floating-point performance since each register can now hold up to four double precision values that can be operated upon in a SIMD fashion. The number of non-destructive operands is also increased from two to three (e.g. $x=x+y$ vs. $x=y+z$) which leads to higher instruction throughput. Intel's latest micro architecture codenamed Haswell maintains the vector register width to 256-bits but provides additional extensions and improvements on the available SIMD instructions in AVX. These new extensions called AVX2 improve floating point performance by a factor of two through Fused-Multiply-Add (FMA) instructions, gather instructions able to load elements using a vector of indices and more advanced shuffling and data rearrangement functionalities [6]. The Intel® Xeon Phi™ coprocessor extends the length of vector registers to 512-bits through a dedicated Vector Processing Unit (VPU). The VPU uses a custom vector instruction set architecture (ISA) different to that of the Xeon processor family and is able to perform SIMD operations on up to eight double precision values. The new vector instructions also provide support for Fused-Multiply-Add (FMA) and gather/scatter operations [7]. A unified SIMD instruction set extension for both Xeon Phi and Xeon processor families is scheduled for implementation in AVX-512 [8]. This will allow for universal 512-bit SIMD support across both architectures and extend the number of available vector registers from sixteen to thirty two. Furthermore, AVX-512 will use a new four byte instruction prefix called *evex* which gives room for both a further increase in the number of SIMD registers and their respective lengths up to 1024-bits.

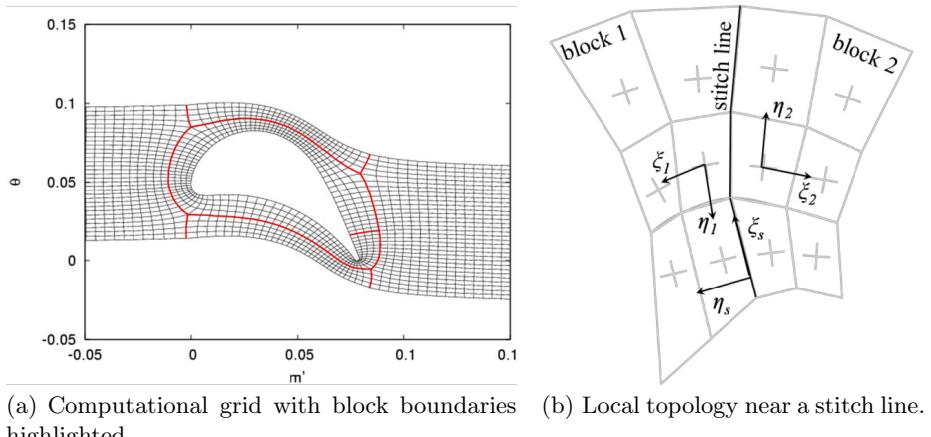
These developments make efficient SIMD execution through code vectorization one of the most promising ways of extracting performance from current and future multi-core and many-core processors. Unsurprisingly, there has been increased interest in enabling SIMD optimisations in scientific applications ranging from generic PDE-based stencil computations [9] to molecular dynamics [2] or Lattice-Boltzmann methods [1]. In CFD, Wang et al. [10] have presented a parallel implementation of a 3D solver for the incompressible Navier-Stokes. Their approach uses SSE and AVX when solving tridiagonal systems and is able to perform backward substitutions within one single SIMD instruction for each row. The authors do not present the speedup derived from vectorization and only describe the operations applied (shuffling) to arrange the matrices in a form fit for SIMD execution. Smith et al. [3] have presented the implementation of a kinetic-theory based vector split Finite Volume Method for the Euler equations, optimised for both vectorization through AVX and threading across the chip using OpenMP. The SIMD optimisation targets the flux reconstruction and state computations and uses AVX intrinsics. The computational domain is decomposed through OpenMP with subsequent decompositions across AVX domains, each holding eight single precision floating point values. The authors report a super-linear speedup of 177x when running on sixteen physical cores compared to single core execution on half a million cells grid. This high increase

in performance is attributed to cache effects although no further explanation is given. The computations are carried out in single precision.

2 Background

2.1 Fast Euler Solver

The test vehicle for this study is a fast Euler solver designed for quick calculation of transonic flow fields past turbine blades. The solver computes inviscid flow solutions in the $m' - \theta$ coordinate system [11]¹. A typical computational domain is shown in Figure 1 and consists of five blocks with different grid sizes as seen in (a). The blocks are connected by stitch lines as illustrated in (b).



(a) Computational grid with block boundaries highlighted

(b) Local topology near a stitch line.

Fig. 1. Computational domain and topology

The Euler equations are solved in discrete form

$$\frac{d}{dt} W_{i,j} \mathbf{U}_{i,j} = \mathbf{F}_{i-1/2,j} - \mathbf{F}_{i+1/2,j} + \mathbf{G}_{i,j-1/2} - \mathbf{G}_{i,j+1/2} + \mathbf{S}_{i,j} = \mathbf{RHS}_{i,j} \quad (1)$$

In equation 1, $W_{i,j}$ is the volume of cell i,j , $\mathbf{U}_{i,j}$ is the vector of conserved variables and the vectors $\mathbf{F}_{i-1/2,j}$, $\mathbf{G}_{i,j-1/2}$ and $\mathbf{S}_{i,j}$ (i -faces) denote fluxes through i -faces, j -faces and source terms, respectively and are evaluated as follows:

¹ θ is the angular position around the annulus. m is the arclength evaluated along a stream surface $dm = \sqrt{dx^2 + dr^2}$ and m' is a normalized (dimensionless) curvilinear coordinate defined from the differential relation $dm' = \frac{dm}{r}$. The $m' - \theta$ system is used in turbomachinery computations because it preserves aerofoil shapes and flow angles.

$$\mathbf{F}_{i-1/2,j} = s_{i-1/2,j}^\zeta \begin{bmatrix} \rho w_\zeta \\ \rho w_\zeta u_m + p n_m^\zeta \\ \rho w_\zeta u_\theta + p n_\theta^\zeta \\ \rho w_\zeta h - p w_\zeta \end{bmatrix}_{i-1/2,j} \quad (2)$$

$$\mathbf{G}_{i,j-1/2} = s_{i,j-1/2}^\eta \begin{bmatrix} \rho w_\eta \\ \rho w_\eta u_m + p n_m^\eta \\ \rho w_\eta u_\theta + p n_\theta^\eta \\ \rho w_\eta h - p w_\eta \end{bmatrix}_{i,j-1/2} \quad (3)$$

$$\mathbf{S}_{i,j} = W_{i,j} \rho_{i,j} \begin{bmatrix} 0 \\ u_\theta^2 \sin \phi \\ u_m u_\theta \cos \phi \\ 0 \end{bmatrix}_{i,j} \quad (4)$$

In the equations above, ζ and η the wise transformed coordinates in the i - and j -directions. The state of the gas is defined by the density ρ , covariant velocities component $u_{m/\theta}$, pressure p and total enthalpy $h = h(\rho, p)$. For the purpose of flux and source term evaluation, the contravariant velocities $w_{\zeta/\eta}$, the normals $n_{m,\theta}^\zeta$ and $n_{m,\theta}^\eta$ and the radial flow angle ϕ are also needed. The physical fluxes are approximated with second order TVD-MUSCL [12] numerical fluxes

$$\begin{aligned} \mathbf{F}_{i-1/2,j}^* = & \frac{1}{2} (\mathbf{F}_{i-1,j} + \mathbf{F}_{i,j}) - \frac{1}{2} \mathbf{R} |\boldsymbol{\Lambda}| \mathbf{L} (\mathbf{U}_{i,j} - \mathbf{U}_{i-1,j}) \\ & - \frac{1}{2} \mathbf{R} |\boldsymbol{\Lambda}| \boldsymbol{\Psi} \mathbf{L} \Delta \mathbf{U}_{i-1/2,j} \end{aligned} \quad (5)$$

The term $\mathbf{R} |\boldsymbol{\Lambda}| \boldsymbol{\Psi} \mathbf{L} \Delta \mathbf{V}_{i-1/2,j}$ represents the second order contribution to the numerical fluxes. $\boldsymbol{\Psi}$ is the limiter and the flux eigenvectors and eigenvalues $\mathbf{R}, \boldsymbol{\Lambda}, \mathbf{L}$ are evaluated at the Roe-average [13] state.

Convergence to a steady state is achieved by a matrix free, implicit algorithm. At each iteration, the correction in the conserved variable vector \mathbf{U} is determined as solution to the linear problem [14]

$$\frac{\delta (W_{i,j} \mathbf{U}_{i,j})}{\delta t} = \mathbf{RHS}_{i,j} + \mathbf{J}_{i,j}^{h,k} \delta \mathbf{U}_{h,k} \quad (6)$$

or, equivalently

$$(W_{i,j} \mathbf{I} - \mathbf{J}_{i,j}^{h,k}) \delta \mathbf{U}_{i,j} = -(\delta W_{i,j}) \mathbf{I} \mathbf{U}_{i,j} + \mathbf{RHS}_{i,j} \quad (7)$$

This problem can be approximated by a problem ruled by a diagonal matrix, if the assembled flux Jacobian $\mathbf{J}_{i,j}^{h,k}$ is replaced with a matrix bearing on the main diagonal the spectral radii $|\tilde{\Lambda}|$ of the flux Jacobian contributions for each cell

$$\mathbf{J}_{i,j}^{h,k} \approx -\text{diag} \left(\sum s |\tilde{\Lambda}|_{\max} \right)_{i,j} \quad (8)$$

At a fixed Courant number $\sigma = \frac{\delta t_{i,j} \sum s |\tilde{\Lambda}|_{\max}}{W_{i,j}}$ this approximation yields the following update

$$\delta \mathbf{U}_{i,j} = \frac{1}{W_{i,j} (1 + \sigma)} (-\mathbf{U}_{i,j} \delta W_{i,j} + \mathbf{RHS}_{i,j}) \quad (9)$$

The solver is implemented as a set of C++ classes. All floating point operations are carried out in double precision. From a computational perspective, the solver can be split into methods operating on face-wise (stencil) and cell-wise loops (non-stencil). The stencil computations can be found in the flux reconstruction phase (iflx) which accounts for more than 65% of the overall runtime. The remaining cycles are spent in non-stencil methods for updating primary (dvar), auxiliary (auxv) and conservative variables (cnsv) and computing the unsteady volume terms (vlhs). In this work, SIMD optimizations are implemented both on stencil and non-stencil kernels.

2.2 Hardware Resources

The Sandy Bridge-EP node configuration used throughout this paper is based on a two socket Intel® Xeon® E5-2650 processor with 64 GB of main memory running Red Hat Enterprise Linux Server 6.4 (Santiago) and kernel version 2.6.32-358.11.1. Each Sandy Bridge chip features eight cores with sixteen available threads through HyperThreading™ clocked at 2.0 GHz and a peak memory bandwidth of 51.2 GB/s. The Ivy Bridge-EP node configuration is based on 2x Intel® Xeon® E5-2650 v2 processors with 32 GB of DDR3 memory clocked at 1600 MHz running CentOS 6.4. Each of the Ivy Bridge CPUs features eight cores capable of running sixteen hardware threads at a clock speed of 2.6 GHz and a maximum memory bandwidth of 59.7 GB/s. Furthermore, the Ivy Bridge-EP chips feature a 20MB L3 cache similar to the Sandy Bridge-EP CPU.

3 Optimisations

3.1 Performance Model

The Roofline [15] performance model will be used in this work. The model targets the discrepancy between processor performance and off-chip memory bandwidth based on a kernel's computational intensity as a unit of operations per byte of main memory traffic (Flops/Bytes). Depending on a kernel's flops per byte ratio, one can use the model to assert whether the computations are compute or memory bound together with an indication of the maximum attainable performance that could be obtained through orthogonal optimisations for that particular kernel on that specific processor. The relation between the compute kernel and selected processor is exhibited in the following formula:

$$\text{Max. GFlops/sec} = \min \left\{ \frac{\text{Peak FP Performance}}{\text{Max. Memory Bandwidth} \times \text{Kernel Flops/Bytes}} \right\} \quad (10)$$

[15] Peak floating point and maximum memory bandwidth are obtained using the micro-benchmarking tools STREAMS and LIKDWID-bench. When aiming for single core optimisations, multi-core bandwidth saturation effects [16] should be taken into account and set accordingly. For example, on Intel's Xeon E5-2650 (Sandy Bridge-EP), a single core's maximum attainable bandwidth when running the STREAM triad benchmark is approximately 13.2 Gflops/sec which

is significantly lower than the 38.5 GFlops/sec obtained when running on four of the eight available cores and which saturated the available memory bandwidth.

3.2 Memory and Data Layout

The first step in optimising the solver is to reduce to loads and stores within computational hotspots. For this reason, storage of *Roe* averages and auxiliary variables is suppressed and replaced by re-computation within stencil updates and flux reconstruction. This leads to a 25% boost in performance. The second step is to apply a number of data layout transformations in order to allow efficient code vectorization. Figure 2 presents the data formats investigated in this paper followed by a brief description of their relevant characteristics. The Array of Structures (AoS) format exhibits very good intra-structure locality as the required elements are contiguously placed in memory however it is inefficient for SIMD computations. Gather and scatter operations are required to transpose the data into a vector register size Structures of Arrays (SoA) format which brings forth performance penalties in current multi-core processors. Due to these considerations, reordering the data to the SoA layout has become common practice in scientific applications wishing to exploit the performance opportunities available through vector computations. There are however specific performance pitfalls when using the SoA approach as well. These are mainly based on inter-structure locality issues as the required elements might reside far away in memory and exceed the virtual page size limit which can result in Transfer-Look-Aside-Buffer (TLB) misses. A compromise between the two can be implemented in the form of a hybrid Array of Structures Structures of Arrays (AoSSoA) which enables an improved intra-structure locality when compared to SoA together with more natural SIMD computations as long as the structure blocks containing groups of identical components are equal or greater in size than the underlying vector register. It is important to note however that modifying an existing application to utilise the hybrid AoSSoA can be a significant task and requires a good degree of encapsulation and prior planning.

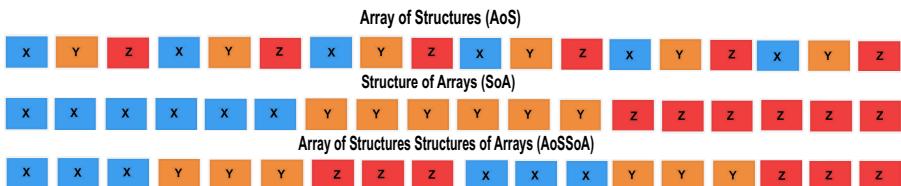


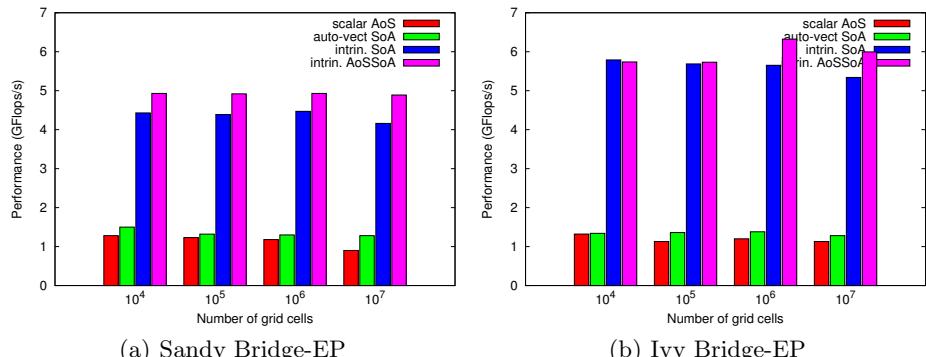
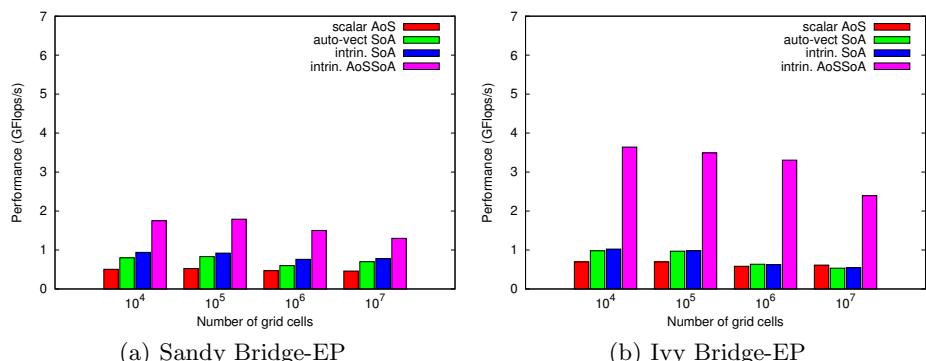
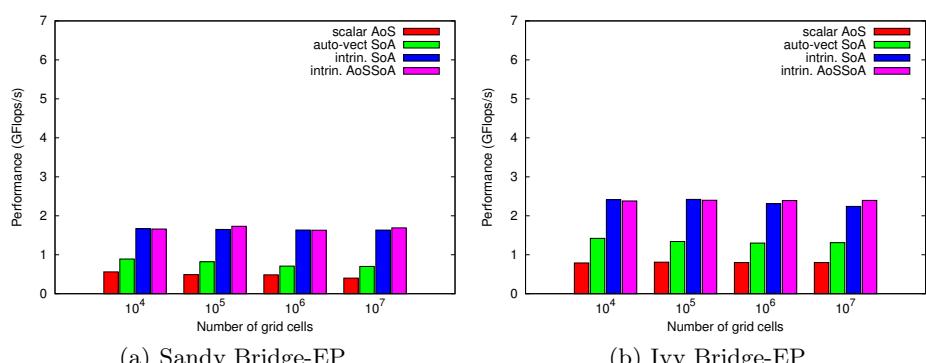
Fig. 2. Array of Structures Data vs. Structures of Arrays Data Layout vs. Array of Structures Structures of Arrays

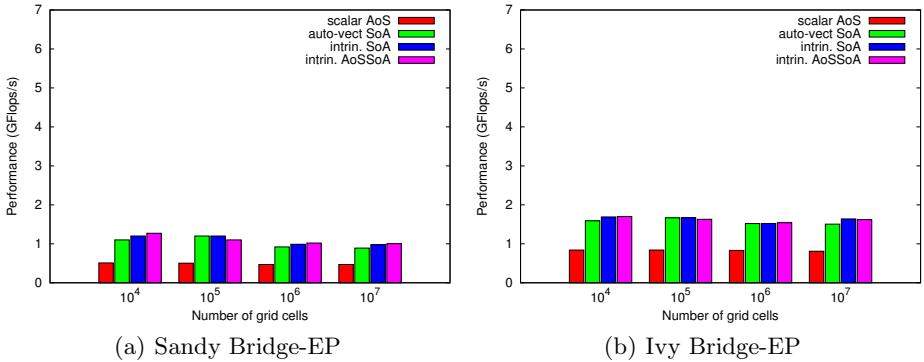
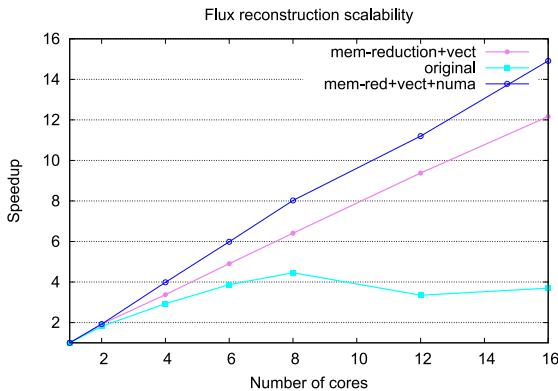
3.3 Vectorization and Shared-Memory

Code vectorization is enabled implicitly using compiler directives and explicitly through intrinsics. For the compiler directives, the `#pragma ivdep` and `#pragma simd` directives are used in order to force SIMDization of selected loops. Full vectorization of stencil computations in the flux reconstruction phase is only possible through intrinsics as the compiler is unable to perform such optimisations due to Read-After-Write and Write-After-Read data hazards. The intrinsics version resolves stream alignment issues through dimension lifted transpositions on the *i*-axis [9]. Shared memory optimisations are focused on alleviating Non-Uniform-Memory-Access (NUMA) effects that can usually propagate in multi-socket systems. The "first touch" technique is utilised where a thread initialises its own section of the working set by "touching it" within a parallel **for loop** which subsequently allows the Operating System to place the respective data chunk within its relative proximity. Domain decomposition is performed within every block in order to avoid load-imbalance due to discrepancies in size. Inter-thread coherence in the flux reconstruction is maintained by coloring the cell boundaries.

4 Results

Figure 3 presents the effect of single-core optimisations of the flux reconstruction kernel. Figures 4-6 show the performance gained from optimising the non-stencil loop-wise kernels. The runs have been performed on grids of different sizes, to explore granularity. The flux reconstruction kernel achieves approximately 70% efficiency on both Sandy Bridge and Ivy Bridge, compared to its roofline model. The non-stencil methods reaches approximately 55%. This would indicate that further single-core optimisations are possible. Profiling reveals the need for code re-factoring as well as algorithmic changes in order to alleviate some of the performance by increasing computational intensity. An attempt has been made to do just that through the utilisation of non-temporal store instructions. This results in palpable benefits in some cases most notably when computing the auxiliary variables as by-passing the caches greatly reduced memory traffic and improved the overall flop per byte ratio. In terms of the data layout transformations, the AoSSoA data structure achieved the highest overall performance when compared to SoA. Subsequent in-depth profiling confirms that this is due to an increase in locality especially when running on sub-blocks the size of a cacheline (eight double precision values). On the fly transpositions from AoS to SoA have not been found to be effective and therefore not pursued in this work. Encapsulation for the AoSSoA format has been done through C++ templates and operator overloading.

**Fig. 3.** Flux reconstruction**Fig. 4.** Conservative variables update**Fig. 5.** Primary variables update

**Fig. 6.** Auxiliary variables update**Fig. 7.** Flux Reconstruction Speedup scaling on Sandy Bridge-EP

5 Conclusions

This paper presents the performance tuning of a 2D multiblock Euler solver through vectorization and shared memory optimisations. Efficient vectorization of stencil methods yields up to a factor of five performance increase on current multi-core architectures whilst shared memory optimisations result in linear scaling across the entire socket contributing to a further 16x speedup. Performance gains are referred to a roofline model for each kernel. A comparison between different data layout transformations and their performance for SIMD computations has also been presented.

Acknowledgements. This work has been supported by EPSRC and Rolls-Royce plc through the Industrial CASE Award 13220161. The authors are indebted to Michael Holiday from Boston Computing for access to Intel® Ivy Bridge-EP nodes.

References

1. Williams, S., Oliker, L., Carter, J., Shalf, J.: Extracting ultra-scale lattice boltzmann performance via hierarchical and distributed auto-tuning. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2011, pp. 55:1–55:12. ACM, New York (2011)
2. Pennycook, S.J., Hughes, C.J., Smelyanskiy, M., Jarvis, S.: Exploring simd for molecular dynamics, using intel xeon processors and intel xeon phi coprocessors. In: Parallel and Distributed Processing Symposium, International, pp. 1085–1097 (2013)
3. Smith, M.R., Liu, J.Y., Kuo, F.A., Wu, J.S.: Hybrid openmp/avx acceleration of a higher order quiet direct simulation method for the euler equations. Procedia Engineering 61, 152–157 (2013), 25th International Conference on Parallel Computational Fluid Dynamics
4. Abel, J., Balasubramanian, K., Bargerion, M., Craver, T., Phlipot, M.: Application tuning for streaming simd extensions. Intel Technology Journal, 1–12 (2009)
5. Gepner, P., Gamayunov, V., Fraser, D.L.: Early performance evaluation of avx for hpc. Procedia Computer Science 4, 452–460 (2011), Proceedings of the International Conference on Computational Science, ICCS 2011
6. Piazza, T., Jiang, H., Hammarlund, P., Singhal, R.: Technology insight: Intel(r) next generation microarchitecture code name haswell. Technical report, Intel Corporation (2012)
7. Zone, I.D.: Intel(r) xeon phi,
<http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-vector-microarchitecture>
 (accessed January 3, 2014)
8. Zone, I.D.: Avx-512 instructions,
<http://software.intel.com/en-us/blogs/2013/avx-512-instructions>
 (accessed April 3, 2014)
9. Henretty, T., Stock, K., Pouchet, L.-N., Franchetti, F., Ramanujam, J., Sadayapan, P.: Data layout transformation for stencil computations on short-vector SIMD architectures. In: Knoop, J. (ed.) CC 2011. LNCS, vol. 6601, pp. 225–245. Springer, Heidelberg (2011)
10. Wang, Y., Baboulin, M., Dongarra, J., Falcou, J., Fraigneau, Y., Maître, O.L.: A parallel solver for incompressible fluid flows. Procedia Computer Science 18, 439–448 (2013)
11. Vavra, M.: Aero-Thermodynamics and Flow in Turbomachines. John Wiley, Los Alamitos (1960)
12. Albada, G., Leer, B., Roberts Jr., W.W.: A comparative study of computational methods in cosmic gas dynamics. In: Hussaini, M., Leer, B., Rosendale, J. (eds.) Upwind and High-Resolution Schemes, pp. 95–103. Springer, Heidelberg (1997)
13. Roe, P.: Approximate riemann solvers, parameter vectors, and difference schemes. Journal of Computational Physics 43(2), 357–372 (1981)
14. Grasso, F., Meola, C.: Handbook of Computational Fluid Mechanics. Academic Press, London (1996)
15. Williams, S., Waterman, A., Patterson, D.: Roofline: An insightful visual performance model for multicore architectures. Commun. ACM 52(4), 65–76 (2009)
16. Treibig, J., Hager, G.: Introducing a performance model for bandwidth-limited loop kernels. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2009, Part I. LNCS, vol. 6067, pp. 615–624. Springer, Heidelberg (2010)

The Performance Characterization of the RSC PetaStream Module

Andrey Semin¹, Egor Druzhinin², Vladimir Mironov², Alexey Shmelev²,
and Alexander Moskovsky²

¹ Intel Corporation, Munich, Germany

andrey.semin@intel.com

² RSC Group, Moscow, Russian Federation

{druzhinin,v.mironov,alexeysh,moskov}@rsc-tech.ru

Abstract. The RSC PetaStream architecture is a massively parallel computer design based on Intel® Xeon® Phi manycore co-processors. Each RSC PetaStream module contains eight Intel Xeon Phi co-processors with PCI-express fabric and Infiniband interconnect for intermodule communication. This paper concentrates on the performance of a single RSC PetaStream module, evaluated with the help of low-level (point-to-point MPI), library (linear algebra, MAGMA) and application-level (classical molecular dynamics, GROMACS and LAMMPS codes) tests. The Intel Xeon E5-2690 top bin CPU dual-socket system has been used for comparison. This early evaluation demonstrates that in general each Xeon Phi co-processor of RSC PetaStream delivers approximately the same performance as dual-socket Intel Xeon E5 system, with only a half energy-to-solution. Fine-grain parallelism of Intel Xeon Phi cores takes advantage of higher messages exchange rates on MPI level for communication of threads placed on different Xeon Phi chips.

1 Introduction

The steady progress in supercomputing field gives rise to 50-100% of peak performance growth every year for the computers on the Top 500 list [1]. With 50 PFlops peak performance of the most powerful supercomputer of the World, this trend suggests the possibility of attaining the level of 1000 PFlops (or 1 exaflops, or 10^{18} flops) by the year 2018-2020. Substantial effort has been spent on investigating potential architecture for exaflop computer [2, 3]. Key issues for a prospective exaflop system were identified in these analyses, with energy efficiency being the most important one. Since individual CPU core performance grew relatively slow (only $\sim x2$) over the last decade, “lighweght” CPU cores of simplified architecture and/or reduced clock frequency are proposed to reduce total power drain and improve energy efficiency. Thus, a new level of parallelism is to be achieved in order to continue supercomputer performance growth. This imposes the following challenges that need to be addressed by system designer:

1. Physical compact placement of computer components to fit existing computer facilities and for better latency.
2. Energy efficiency on all levels, taking into account that exaflop system might consume tens of megawatts of power
3. Resilience to the failure of supercomputer individual components, such as nodes.
4. Protecting investment in software, since modern simulation codes usually have history of dozens of years of development by generations of researchers and cannot be easily rewritten from the scratch.

RSC PetaStream architecture addresses these issues by conceptual and physical design. The liquid cooling in PetaStream gives not only energy efficiency, but also provides compactness and density as well. The x86 compatibility of Intel Xeon Phi eases porting efforts and gives better chance that software developed for this platform will survive next 5 years. Resiliency is an open issue, but overall node reliability benefits from lower operation temperature, secured by liquid cooling technology. As well, fault tolerance can be addressed with incremental checkpoint, supported by local SSD storage.

The working sample of the RSC PetaStream has been constructed by RSC Group and used for tests, is presented in this paper. RSC PetaStream hardware is sketchily described in the next chapter, while benchmark methods are described in the Chapter 3, and performance measurements are presented and discussed in the next one. Related work is described in Chapter 5.

2 RSC PetaStream Architecture

RSC PetaStream is an innovative implementation of massively parallel coprocessor-oriented architecture. It relies on the power of novel Intel Many Integrated Core (MIC) architecture. Compute nodes are based on Intel® Xeon Phi 5120D coprocessor featuring 60 x86-compatible cores and 8GB of high-bandwidth (5.5 GT/s) GDDR5 memory, providing 1 TFlops peak performance per node. To ensure maximum density, high I/O throughput, reliability and manageability multiple nodes inside RSC PetaStream system are grouped into a module that provides liquid cooling, highly efficient power delivery and conversion, and mechanical assembly of the nodes into a computing rack. Every compute module contains eight compute nodes combined in one compact enclosure, which gives total 8 TFlops theoretical performance per compute module. Nodes in such a module form a homogeneous compute environment with fast PCIe/Infiniband interconnect. 128 PetaStream modules can be integrated into double-sided 100 by 100 cm 42U tall cabinet, which aggregates total peak performance of 1 PFlops.

RSC PetaStream compute module is based on single-socket Xeon E5-2697v2 CPU server as a host computer. Module may have up to 128 GB of DDR3R-1866 main memory and up to three expansion cards. Two PCI Express switches are used to provide communication between four compute nodes as well as dual-port Mellanox Connect-IB Infiniband card each (see Fig. 1). As a result, RSC PetaStream module has 4x FDR Infiniband ports (more than 200 Gbps bandwidth aggregated), connected to the

external Mellanox FDR MSX6025F switches by fiber optic cables. The inter-module connection topology can be flexibly configured depending on the user demands. The module can have up to 5 local SSDs for data storage with aggregate capacity up to 0.8TB/4TB raw configurable into RAID 0, 1, 10 or 5. The host's server board (Intel Server Board S1600JP4) is equipped with 4x Gigabit Ethernet (RJ45) ports, which can be used for management and monitoring networks, featuring full IPMIv2 with KVM over IP support.

Energy efficiency is secured by several design decisions. First one is a power distribution subsystem, which is based on 400V DC power supply standard and is designed for efficiency, redundancy and compactness. Second one is a contact liquid cooling technology that is used to remove all the heat from the module (up to 2.7 kW). It is extremely efficient: only 6% of total power is used for system cooling. Finally, power monitoring and capping capabilities are enabled for every Xeon Phi co-processor of PetaStream.

Each node runs its own image of operating system (Linux-based OS provided as part of Intel MPSS), and Linux OS is run on host (see Table 1 for details). Due to majority of PetaStream computation power comes from Xeon Phi chips, it does make sense to run application on Intel Xeon Phi cards, and use the host's CPU for support and service functions. Thus, application is run on uniform field of Xeon Phi nodes – at least one MPI rank per node – compatible with “native” mode for Xeon Phi. At the same time, it is important to note that offload model is also supported by RSC PetaStream architecture, and, for example MAGMA benchmarks were run exactly in the offload mode. Application software for RSC PetaStream platform can be developed with common toolkit from Intel, Cluster Studio XE 2013, no modifications has been made in the MPSS software stack to preserve compatibility with traditional Xeon Phi based solutions. Intel MPI 4.1, as well as Intel MPSS 2.1 software stack are present. RSC PetaStream supports all programming models suggested by Intel for MIC architecture, namely native MIC programming, symmetric mode and offload mode. RSC PetaStream system leverages Intel Node Manager Technology to control and monitor node power consumption of every node, that mechanism can be used to implement flexible power energy and optimization strategies to help HPC sites save power and reduce operational costs.

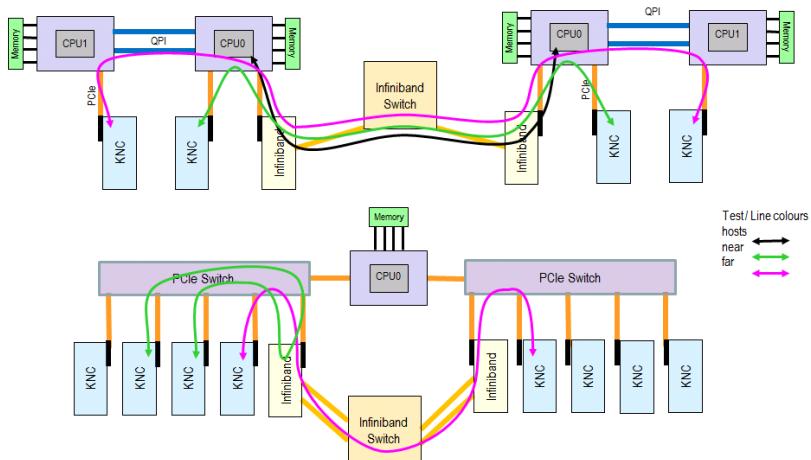
3 Benchmarks

This study is focused on the performance of a single compute module only. A set of tests was performed for the first evaluation of PetaStream hardware. They include low-level performance measurements, set of linear algebra tests as well as application-level tests.

PetaStream results were compared with those obtained on the RSC Tornado Intel Xeon Phi-equipped platform; the same has been used for MVS-10P supercomputer, which is installed at Joint Supercomputer Center of Russian Academy of Sciences. It is based on dual-CPU Intel Xeon E5-2600 nodes with 2x Intel Xeon Phi modules. Their detailed configurations are summarized in Table 1.

Table 1. Configurations of the test systems

	RSC PetaStream	RSC Tornado
Host processors	1x Xeon E5-2697v2	2x Intel Xeon E5-2690 (C2-step)
Co-processor	8x Xeon Phi 5120D (C0-step)	2x Xeon Phi SE10X (B1-step)
RAM amount/speed	64GB DDR3R-1600	64GB DDR3R-1600
Main board	Intel Server Board S1600JP	Intel Server Board S2600JFF
Main board BIOS	02.01	01.06
PM settings	cpufreq and PC6 enabled	EIST and Turbo enabled
Infiniband HCA	Connect-IB, 2-port	ConnectX-3 on-board
Host OS	CentOS 6.4	CentOS 6.2
MPSS	2.1 Gold Update 3 (6720-21)	2.1 Gold Update 2 (5889-14)
OFED version	3.5-rc3	1.5.4.1
Infiniband switch	Mellanox FDR MSX6025F. 1 hop between hosts	

**Fig. 1.** “Near” and “far” communication patterns in RSC Tornado (top) and RSC PetaStream (bottom)

3.1 OSU MPI P2P Benchmarks

Several MPI OSU point-to-point benchmarks (OSU MPI microbenchmarks v4.2 [4]) were used to measure the performance of the intra-node communication. While large share of all application software uses MPI API to enable communication among its processes, MPI primitives are usually well optimized and can characterize underlying hardware as well. In current study bandwidth, latency and message rate between compute nodes were studied. In first two tests we measure the connection performance relative to the message size between two processes located on different compute nodes (either hosts or co-processors). In the last test we measure the dependence of message rate (1K messages) from the number of communicating pairs of ranks, located on different compute nodes.

Both studied systems are inhomogeneous with respect to MIC-to MIC communication. On the RSC Tornado platform two MICs and Infiniband adapter are plugged to the PCIe bus directly. To ensure maximum CPU-MIC I/O throughput, co-processors are tied to different CPUs. In this case process-to-process messages between MICs on single node passes through the QPI interconnect which links the CPUs. In case of MIC-to-MIC inter-node communication one can distinguish two types of route (“near” and “far”, see Fig. 1), on which Infiniband adapters and peers are located in the same or different PCIe branches respectively. In the latter case additional QPI overhead influence the overall performance. In current work both patterns has been studied as well as overhead of host-host communication (“host” on Fig. 1).

In RSC PetaStream Xeon Phi modules and Infiniband cards are connected through the PCIe switch that eliminates the necessity of QPI communication. They are two possible communication routes, which we also call “near” and “far” for consistency (see Fig. 1).

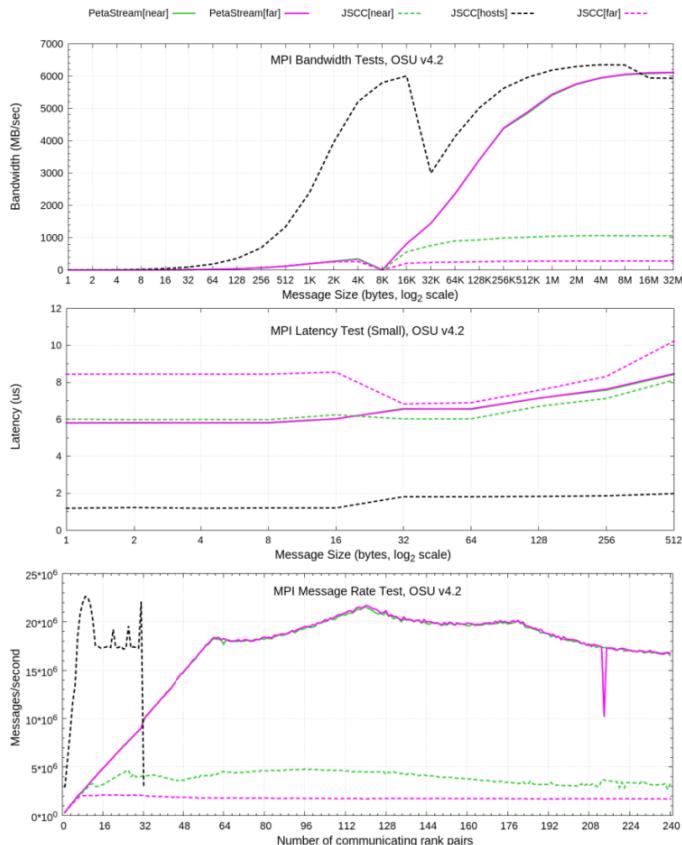


Fig. 2. Point-to-point bandwidth (top), latency on small messages (middle) and MPI message rate (bottom)

3.2 Linear Algebra Libraries

MAGMA [5, 6] is a novel linear algebra package that is designed to run efficiency on multi-core architectures, developed at Innovative Computing Laboratory, at University of Tennessee, Knoxville, Tennessee. The MAGMA MIC [7] v. 1.1 has been used in current study. MAGMA MIC doesn't rely on MPI for communication but on SCIF APIs. We have utilized standard benchmark suite, provided by library developers. Cholesky, LU and QR double precision tests has been used, varying problem size ($N \times N$ matrices, $N=1000 \div 65000$) and number of compute nodes (from 1 to 8 Intel Xeon Phi nodes).

3.3 Application-Level Tests

LAMMPS [8] is a popular open-source molecular dynamics package commonly used in material science and computational biology. Standard Lennard-Johns liquid benchmark provided by code developers was used to study LAMMPS performance in weak and strong scaling regime. The performance was measured in the CPU cost per atom per timestep as it suggested by code developers.

GROMACS [9] is widely adopted molecular dynamics software with hundreds of citations every year. It is primarily designed for large-scale simulation of systems containing up to several millions of atoms on multicore and multiprocessor environments and highly optimized for performance. The performance has been measured using workload of water (1.536 million atoms). Energy-to-solution data has been gathered for GROMACS, using the Intel Node Manager capabilities to measure platform power consumption for the both RSC PetaStream and RSC Tornado systems.

Both LAMMPS (ver. 11 Mar 2013) and GROMACS (ver. 4.6) had been compiled using Intel Compiler 14.0.0, Intel MKL 11.1.1.106 and Intel MPI 4.1.1.036.

4 Performance Results

4.1 Low-Level Bandwidth Benchmark

We have summarized performance data of OSU MPI benchmarks on Fig. 2. The maximum bandwidth (Fig. 2, top) for PetaStream is on par with host-to-host communication for Intel Xeon E5-2690 based system. However, latency (Fig. 2, middle) on small messages is substantially better for CPU-to-CPU communication, we attribute that to higher clock frequency of CPU comparing to the Xeon Phi. The communication between processes on Xeon Phi is dramatically improved on RSC PetaStream comparing to the Tornado platform. The maximum bandwidth available has risen from 1 GByte/sec (near) to almost 6 GByte/sec, message rate (Fig. 2, bottom) has improved from 5 million messages per second to more than 20 million, with approximately same latency. We attribute that to more efficient communication intermodule network of the RSC PetaStream for the both near and far communication. Message rate exhibits linear growth upon number of communicating pairs until it reaches a bottleneck. In JSCC Tornado system it is either a PCIe controller on CPU ("near"-type interconnect) or a

QPI interface (“far”-type interconnect). In PetaStream “near” and “far”-type interconnects has similar message rate approaching “host”-type results.

It should be stressed, that results are measured for node-to-node communication over FDR Infiniband interconnect. The nodes in different modules are connected the same way, so the presented results will remain the same for intra-module communication. Thus, uniform net of nodes is formed with RSC PetaStream, even when multiple modules are connected.

4.2 Linear Algebra

MAGMA MIC shows a fairly good performance results on RSC PetaStream. Single RSC PetaStream compute node performance approaches 0.76 TFlop/s in double precision LU benchmark (Fig. 3) that is twice more powerful than dual-socket Intel Xeon E5-2690 server. On eight compute nodes it reaches 5.1 TFlop/s for LU benchmark, which is 63% of theoretical performance. The performance of Cholesky and QR decomposition is less, and additional nodes contribute less performance, but the similar trends were observed. The maximum observed performance values for these routines are respectively 3 TFlop/s and 3.8 TFlop/s per compute module. It should be noted, that higher on-board memory for Intel Xeon Phi-based nodes could produce higher results.

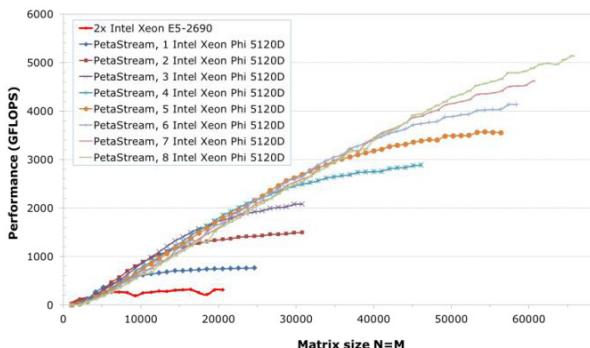


Fig. 3. MAGMA MIC LU factorization performance on RSC PetaStream

4.3 Application Level Benchmarks

LAMMPS performance results are presented on Fig. 4. LAMMPS package shows close-to-linear weak and strong scaling in this benchmark both for RSC PetaStream and CPU-based architectures. In this test, a single RSC PetaStream compute node performance is almost equivalent to dual-socket Intel Xeon E5-2690 server. GROMACS strong scaling is slightly worse most possibly due to accounting of long-range electrostatic interactions in this benchmark. The single RSC PetaStream compute node performance is again close to those of 2x Intel Xeon E5-2690 CPUs, but

GROMACS scales better on PetaStream than on E5-based system. The energy-to-solution metric, that has been measured for GROMACS, demonstrates, that RSC PetaStream requires twice less energy (~0.7 kWh) than RSC Tornado (~1.4 kWh), which is a good proof of PetaStream energy efficiency in real-life applications.

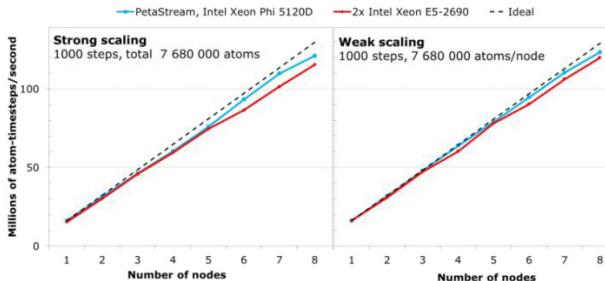


Fig. 4. LAMMPS strong (left) and weak (right) scaling on PetaStream. Each Xeon E5 and Xeon Phi node has 16 and 240 threads running respectively.

5 Related Work

Multicore architectures quickly have been recognized for their computing potential and became an integral part of modern supercomputers. This work is devoted to the characterization of novel HPC platform RSC PetaStream, which computing power is purely based on Intel MIC technology. We do not rely on any standard performance evaluation suite, but use a mixture of low-level performance tests (OSU benchmarks for MPI) as well as popular linear algebra libraries (LAPACK, MAGMA) and software applications benchmarks (LAMMPS, GROMACS) to investigate various aspects of platform performance.

The low-level results of OSU MPI microbenchmark show that RSC PetaStream node communication performance is on par with modern supercomputer architectures (see e.g. [10]). It is not unexpected: Infiniband FDR interconnect, which is used in the RSC PetaStream architecture, is widespread among the top-level supercomputers from the Top500 list. However, use of coprocessors means the heterogeneity often introducing many new communication planes with different performance overhead [11].

Linear algebra is extensively harnessed by many scientific applications and has a very high potential parallelism. Therefore it is a must benchmark for any performance test suite. The MAGMA linear algebra package is frequently used to benchmark GPU performance. Its specialized version for Intel MIC architecture was used in this study to measure linear algebra performance of RSC PetaStream architecture. RSC PetaStream significantly (at least two times) surpasses in performance Nvidia Tesla M-Class GPUs (~300 GFlop/s per chip, see [12]) and is compatible with modern Nvidia Tesla K20x (~1000 GFlop/s per chip, see [6]). In contrast to MAGMA, LINPACK [13] is a general purpose linear algebra package, which is available for all architectures. The LINPACK maximum performance on RSC PetaStream module (5.6 TFlop/s per compute module) is close to MAGMA's result and is very promising,

delivering the record computational density. For example, the performance of comparable in size Cray XK7 module is several times lower [1, 14].

The performance of LAMMPS and GROMACS molecular dynamics packages on RSC PetaStream platform is fairly good in comparison to other HPC systems [15–17], including GPU-based [17, 18]. Generally, the MD benchmarks demonstrate lower number of GFlop/s than linear algebra. However, each RSC PetaStream compute node equal or even outperforms a dual-socket Intel Xeon 2600 computer and its power consumption is two times lower than the latter. One RSC PetaStream compute node is also faster than one Cray XK6 node equipped with 16-core AMD Opteron 6200 and NVIDIA Tesla X2090 [17]. Thus, RSC PetaStream architecture suits well for molecular dynamics calculations in spite of the fact that they don't reveal its full computing power.

DEEP project [19] is aimed to develop similar type of architecture for its “booster” nodes. However, final performance results were not available publicly at the time of writing this paper.

6 Conclusion

We have briefly described RSC PetaStream architecture and early performance evaluation results. The RSC PetaStream with 8 Intel Xeon Phi's in compact form-factor has been evaluated with low-level, library and application level benchmarks. On low lever, point-to-point communication bandwidth (6 GBytes/sec) of RSC PetaStream is equal to attainable on intra-node communication of CPU based clusters with FDR Infiniband interconnect. The message rate ($2 \cdot 10^7$) gives a room to implement latency hiding techniques to alleviate higher communication latency. Since these results are demonstrated for node-to-node communication over FDR Infiniband interconnect, presented figures will stand correct for inter-module communication.

Linear algebra benchmarks with MAGMA MIC 1.1 library demonstrate good share of theoretical performance (63% or 5.1 TFlop/s on LU test). With total power consumption of RSC PetaStream module of 2.1 kW, this gives 2426 MFlops/Watt – better than #10 of Green500 list of November 2013 [20].

The RSC PetaStream performance measurements for molecular dynamics codes LAMMPS and GROMACS demonstrate that, in general, one RSC PetaStream node may be equivalent to dual-socket Intel Xeon E5-2690 node in the speed of computation. Energy-to-solution for RSC PetaStream is almost twice as better than for computation on Intel Xeon E5-2690 CPU.

The results above support RSC PetaStream competitiveness to other High-performance computing platforms it terms of performance and energy efficiency.

Acknowledgements. We thank Jack Dongarra, Stanimire Tomov and the team of the Innovative Computing Laboratory of the University of Tennessee for the invaluable help in testing of MAGMA package on RSC PetaStream.

References

1. TOP500 Supercomputer Site, <http://www.top500.org>
2. Kogge, P., Bergman, K., Borkar, S., et al.: ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. Technical report. Gov. Procure. TR-2008-13, 278 (2008)
3. Kogge, P.: The Challenges of Petascale Architectures. *Comput. Sci. Eng.* 11, 10–16 (2009)
4. OSU MPI benchmarks, <http://mvapich.cse.ohio-state.edu/benchmarks>
5. Agullo, E., Demmel, J., Dongarra, J., et al.: Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *J. Phys. Conf. Ser.* 180, 012037 (2009)
6. Dongarra, J., Dong, T., Gates, M., et al.: MAGMA: Matrix Algebra on GPU and Multicore Architectures. In: SC12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. IEEE Computer Society Press, Salt Lake City (2012)
7. Dongarra, J., Gates, M., Jia, Y., Kabir, K., Luszczek, P., Tomov, S.: MAGMA MIC: Linear Algebra Library for Intel Xeon Phi Coprocessors, http://icl.cs.utk.edu/projectsfiles/magma/pubs/24-MAGMA_MIC_03.pdf
8. Plimpton, S.: Fast Parallel Algorithms for Short-Range Molecular Dynamics. *J. Comput. Phys.* 117, 1–19 (1995)
9. Hess, B., Kutzner, C., van der Spoel, D., Lindahl, E.: GROMACS 4: Algorithms for Highly Efficient, Load-Balanced, and Scalable Molecular Simulation. *J. Chem. Theory Comput.* 4, 435–447 (2008)
10. Kerbyson, D.J., Barker, K.J., Vishnu, A., Hoisie, A.: A performance comparison of current HPC systems: Blue Gene/Q, Cray XE6 and InfiniBand systems. *Futur. Gener. Comput. Syst.* 30, 291–304 (2014)
11. Kandalla, K., Venkatesh, A., Hamidouche, K., et al.: Designing Optimized MPI Broadcast and Allreduce for Many Integrated Core (MIC) InfiniBand Clusters. In: 2013 IEEE 21st Annual Symposium on High-Performance Interconnects, San Jose, CA, USA, pp. 63–70 (2013)
12. Yamazaki, I., Tomov, S., Dongarra, J.: One-sided Dense Matrix Factorizations on a Multicore with Multiple GPU Accelerators. *Procedia Comput. Sci.* 9, 37–46 (2012)
13. Petitet, A., Whaley, R.C., Dongarra, J., Cleary, A.: HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers (2008), <http://www.netlib.org/benchmark/hpl>
14. Dongarra, J.: Performance of Various Computers Using Standard Linear Equations Software (Linpack Benchmark Report). Technical report (2013)
15. You, H., Lu, C.-D., Zhao, Z., Xing, F.: Optimizing utilization across XSEDE platforms. In: Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery, XSEDE 2013, p. 1. ACM Press, New York (2013)
16. Loeffler, H., Winna, M.: Large biomolecular simulation on HPC platforms III. AMBER, CHARMM, GROMACS, LAMMPS and NAMD, Warrington, UK (2012)
17. LAMMPS Benchmarks, <http://lammps.sandia.gov/bench.html>
18. Nvidia Corporation: GROMACS 4.6 Pre-Beta Benchmark Report, Revision 1.0 (September 10, 2012), <http://www.nvidia.com/docs/IO/122634/gromacs-benchmark.pdf>
19. Eicker, N., Lippert, T., Moschny, T., Suarez, E.: The DEEP project: Pursuing cluster-computing in the many-core era. In: Proc. of the 42nd International Conference on Parallel Processing Workshops (ICPPW) 2013, Workshop on Heterogeneous and Unconventional Cluster Architectures and Applications (HUCAA), Lyon, France, pp. 885–892 (2013)
20. Green500 list (November 2013), <http://green500.org/lists/green201311>

Deploying Darter - A Cray XC30 System

Mark R. Fahey*, Reuben Budiardja, Lonnie Crosby, and Stephen McNally

University of Tennessee, Knoxville

Joint Institute for Computational Sciences, Knoxville, TN, USA

{mfahey, reubendb, lcrosby1, smcnally}@utk.edu

Abstract. The University of Tennessee, Knoxville acquired a Cray XC30 supercomputer, called Darter, with a peak performance of 248.9 Teraflops. Darter was deployed in late March of 2013 with a very aggressive production timeline - the system was deployed, accepted, and placed into production in only 2 weeks. The Spring Experiment for the Center for Analysis and Prediction of Storms (CAPS) largely drove the accelerated timeline, as the experiment was scheduled to start in mid-April. The Consortium for Advanced Simulation of Light Water Reactors (CASL) project also needed access and was able to meet their tight deadlines on the newly acquired XC30. Darter's accelerated deployment and operations schedule resulted in substantial scientific impacts within the research community as well as immediate real-world impacts such as early severe tornado warnings [1].

Keywords: Cray XC30, Aries interconnect, Hyperthreading, rank placement, storm prediction, light water reactors, astrophysics.

1 Overview

In order to meet the ever increasing computational science research needs at the University of Tennessee, Knoxville (UTK) and its collaborating institutions, the Joint Institute for Computational Sciences (JICS) acquired a new Cray XC30 supercomputer, called Darter¹, with a peak performance of 248.9 Teraflops. As computational techniques become increasingly important to a rising number of research fields, interest in maintaining computational resources for research groups has grown across campus. The XC30 is Cray's latest supercomputer designed to deliver sustained performance by combining the Intel® Xeon™ processors, the next-generation Aries™ interconnect, Dragonfly network topology, integrated storage solutions, and enhancements to the Cray operating system and programming environments.

Darter was deployed in late March of 2013 with a very aggressive production timeline. The system was deployed, accepted, and placed into production in only 2 weeks. The Spring Experiment for the Center for Analysis and Prediction of

* Joint Faculty with the Industrial and System Engineering Department.

¹ Named after the Snail Darter fish that is found in East Tennessee freshwater in the United States.

Storms largely drove the accelerated timeline, as the experiment was scheduled to start in mid-April; see Section 3.2. JICS was able to effectively utilize staff expertise to achieve this remarkable feat. Scientists from CAPS immediately began functionality testing as they worked to port their scientific models to this new computing architecture. The consistency of the Cray multigenerational software stack and proven hardware integrity also played important roles in the successful Darter deployment.

The next section describes the processors, interconnect, storage and software. The acceptance test (3.1) section briefly describes performance results on common codes and kernels, while the science highlights (3.2, 3.3, and 3.4) sections detail the easy nature of application porting, which ultimately has resulted in increased productivity for scientific research.

2 Architecture

Darter is a resource with a completely new architecture compared to earlier generations of Cray machines: Intel processors replacing AMD, new network topology (dragonfly vs. torus), and new interconnect (Aries versus Gemini and Seastar). Yet from the user's perspective there's little to no added complexity to realize this performance boost as applications are easy to port. Although somewhat contrary to recent trends (the need to exploit more parallelism and platform specific tunings) this inherent performance increase is appreciated and well accepted among the user community. In addition, system deployment proved to be straightforward and quick given that Cray continues to build off their successful system implementation model. Furthermore, system integration and acceptance testing were successfully completed in an extremely short timeframe (7 days.)

Darter is comprised of 4 compute cabinets, 5 cooling cabinets, and 1 management cabinet. In total, there are 187 compute blades each made up of 4 compute nodes and 5 service blades that contain 2 service nodes per blade totaling 192 physical blades. Therefore, there are 748 compute nodes and 10 internal service nodes. Darter is also configured with 2 external login nodes. Table 1 shows a node configuration summary.

Table 1. Darter node configuration

Darter node architecture	
processor	Intel Xeon E5
cores/processor	8
hyperthreads/core	2
core frequency	2.6 GHz
sockets/node	2
memory/node	32 GB
mem bandwidth/node	25.6 GB/s
total number of nodes	748
total number of processors	1,496
total number of cores	11,968
network card	Aries

Processors

Darter has Intel Xeon E5-2670 processors. These processors have 8 physical cores with a clockspeed of 2.6 GHz. Note these processors have 16 threads - Intel's

Hyper Threading Technology. Hyper-Threading refers to the use of a single physical processor as two logical processors [2]. Tests show that the performance gain with Hyper-Threading varies and can even be negative across different applications and numerical kernels [3,4]. With the launch of the XC30 with Intel processors, Cray decided to support the use of Intel Hyper-Threading by default, although a runtime option is necessary for its utilization. This provides scientific applications the ability to explore the utilization of this technology, particularly in the context of a scalable supercomputing architecture.

Interconnect

The XC30 network is based on a high-bandwidth, low diameter topology called Dragonfly. The Dragonfly network topology is constructed from a configurable mix of backplane, copper, and optical links. The XC30 implements the Dragonfly network with the Aries interconnect ASIC. Each compute blade contains 4 Aries NICs and 1 Aries interconnect router chip. Each Aries NIC is mapped to an individual compute node and the 4 onboard Aries NICs are connected to the Aries router chip. Each Aries ASIC communicates via a standard PCI Express Gen3 $\times 16$ host interface to each of the 4 compute nodes on that blade. Network requests are issued from the compute node to the NIC over the PCI bus. Network packets are typically routed adaptively, on a packet-by-packet basis, through the Aries network and therefore can leverage both minimal and non-minimal routing paths. Within adaptive routing 4 randomly selected routes (2 minimal and 2 non-minimal) are selected and evaluated using up-to-date network information. The system will determine which path is best based on load information evaluation and the path with the lightest load is selected. With the concept of adaptive routing, the Aries interconnect is also much more fault tolerant than earlier predecessors such as the SeaStar interconnect. Link and node failures do not debilitate the system as they once did. The Hardware Supervisory System (HSS) performs the task of routing around faulty components, which provides warm swap functionality for most hardware failures [5].

Storage

The JICS XC30 was acquired with an attached Sonexion filesystem. This filesystem consists of 2 Sonexion C1600 Scalable Storage Units (SSU) each with 82 3TB disks, providing 334 TB of usable space and a peak of 12 GB/s to the system. Darter is connected to the filesystem via 4 I/O nodes. These I/O nodes are lustre network (LNET) [6] routers directly connected to the Sonexion through a single FDR infiniband cable per node. During acceptance, IOR tests achieved write rates as high as 11.60 GB/s and read rates as high as 7.09 GB/s. The Sonexion system is intended to serve as a storage appliance, as all of the associated infrastructure is self-contained within the storage solution. Along with the 2 SSUs, the storage solution also provides 4 management nodes in a single Metadata Management Unit (MMU). Of the 4 management nodes, the system has 2 Cray Sonexion System Management (CSSM) nodes in a high availability pair, 1 Metadata Server (MDS), and 1 Lustre Management Server (MGS).

System Software

Cray XC30 systems utilize the Cray Linux Environment (CLE), which is a Linux operating system based on SUSE distributions designed to run large complex applications and scale efficiently to more than 500,000 processor cores. The CLE features a lightweight compute kernel that includes support for POSIX system calls, POSIX threads, programming models, application networking, Cray provided system administration utilities, performance tools, and job management tools [7]. Darter leverages many of the provided CLE features as well as expands on some areas, such as job management. Darter’s batch system is a combination of the Torque resource manager and Moab scheduler that is used on all other JICS compute resources. Darter is running Torque 4.1.x and Moab 7.1.x, which provides direct communication between Torque and ALPS that allows for more flexibility in externalizing the batch system so that users can still interact with the batch system even if the compute system is down for maintenance. Darter also provides support for automatic library tracking with the Automatic Library Tracking Database (ALTD) [8]. On Darter’s external login nodes (ES-Login) Cray’s ESL is leveraged to provide communication between the compute system and the ESLogin nodes. The ESL software presents the ESLogin nodes as internal login nodes to users so that their workflows remain the same even when the compute system is down.

Programming Environment

Darter shares a similar programming environment as other previously deployed Cray systems such as the XK/XE line and the venerable XT line. This benefits users that are already familiar with previously available Cray systems in porting and running their applications on Darter. Since the programming environment is nearly architecture agnostic, porting an application previously built on another Cray system to Darter is often as easy as recompiling and relinking while maintaining the same Makefile or other build configuration.

Cray’s programming environment uses Cray’s compiler wrapper `f77`, `CC`, `cc` to compile and link with Fortran, C++, and C compilers, respectively. The compiler wrappers call the underlying “real” compiler while taking care of supplying both include and libraries paths for Cray’s MPI and scientific libraries behind the scene. On Darter, the supported compilers are the Cray Compiling Environment (CCE), the Intel Compilers, and the GNU Compiler Collection (GCC).

3 Applications and Workloads

3.1 Acceptance Testing

All production JICS resources are put through a defined set of acceptance tests to determine that their functionality and performance baselines meet expected parameters. The acceptance tests are divided into hardware and application tests. The hardware tests include initial diagnostics and testing that confirm the delivered hardware and firmware are correctly configured and functioning. The application tests are subdivided into functionality, performance, and stability

phases. At the end of each phase the application acceptance tests are continuously run on the system for a specified amount of time. This “continuous system integration” step ensures that all parts of the system are tested under conditions similar to production utilization.

A series of benchmarks including HPL [9], MPI-Stream, G-FFTE [10], Intel MPI Benchmark (IMB), and IOR [11] were utilized to test aspects of the system’s processors, memory, interconnect, and filesystem for proper performance. The HPL acceptance test obtained 200.1 Teraflops on 744 nodes, which is equivalent to 80.8% of the theoretical peak of the machine. The MPI-Stream test, run on 675 nodes, measured an aggregate of 52,308.3 GB/s or 77.49 GB/s, which is 75.7% of peak. The G-FFTE test on 8192 cores obtained 1.992 Teraflops. Additionally, scientific applications (PARATEC and WRF [12]) and the HPCC [10] benchmarks were run on the system to test for proper system operation. These applications and benchmarks along with IMB were run continuously during the stability test phase of the application acceptance test. Over 700 tests were executed obtaining a pass percentage of over 99.8%.

Comparing this acceptance test with previous testing on a Cray XT5 (Kraken), it is clear that there are significant improvements in application performance due to the improvements in processor, memory, and interconnect performance. The Intel Xeon processors provide a factor of two improvement in both peak performance and the Highly Parallel Linpack (HPL) benchmark on a per-core basis over the AMD® Operton™ processors utilized in Kraken. The additional improvements in memory bandwidth provide another three-fold increase in performance over the XT5. A set of six applications including WRF [12] were run between these systems showing average per core performance improvements by factors between 1 and 4 [13]. WRF showed the most improvement (nearly 4 times) while all but one additional applications demonstrated per core performance improvements nearly equal to or greater than 2.

3.2 Storm Prediction

The Center for Analysis and Prediction of Storms (CAPS) at the University of Oklahoma ran Storm-Scale Ensemble Forecasts (SSEF) on Darter, in supporting the National Oceanic and Atmospheric Administration (NOAA) Hazardous Weather Testbed (HWT) 2013 Spring Experiment. The NOAA HWT Spring Experiment, with participants including NOAA Storm Prediction Center (SPC), the National Severe Storm Laboratory (NSSL), and CAPS, is a yearly high profile experiment that investigates the use of convection-allowing model forecasts as guidance for the prediction of hazardous convective weather. A variety of model output is examined and evaluated daily and experimental forecasts are created and verified to test the applicability of cutting-edge tools in a simulated forecasting environment. Since its beginning in 2007, CAPS SSEF is the most important model forecast component to HWT.

The 2013 CAPS SSEF runs lasted seven weeks between April 22 and June 7, 2013, overlapping with the HWT 2013 Spring Experiment from May 6 to June 7, 2013. During the 2013 Spring Experiment, a thirty-member storm-scale

ensemble forecast of 48 hours, at 4-km horizontal grid spacing over the entire continental United States (CONUS), was produced daily starting at 00 UTC. Three state-of-the-science numerical weather prediction (NWP) models were used. They are the Advanced Research version of the Weather Research and Forecast model (WRF-ARW), the Advanced Regional Prediction System (ARPS), and the Navy COAMPS model system. Each ensemble member forecast has unique initial condition and lateral condition perturbations and model physics options. Over 140 WSR-88D Doppler weather radar and conventional observations over the CONUS were assimilated in real-time to each member using the ARPS 3DVAR and Complex Cloud Analysis package. Ensemble forecast products, including probabilistic severe weather guidance, tornadic weather potential and intensity, flash flood and damaging wind guidance, were made available to HWT participants 12-36 hours in advance.

The dedicated use of Darter made a big contribution to a very successful 2013 Spring Experiment season. It allowed CAPS to not only produce realtime baseline storm-scale ensemble forecasts for HWT, but to also run several combinations of ensemble configuration in order to find optimization for more accurate prediction of severe weather events. The CAPS SSEF did very well in capturing major hazardous weather events throughout the season, including the devastating May 20 Moore, Oklahoma Tornado case. Fig. 1 shows the 20 hour SSEF forecast of that day, valid around the time of initial tornado touch down at Moore, Oklahoma. This research has led to substantial scientific impacts within the research community as well as immediate real-world impacts such as early warnings for the May 20 Moore, Oklahoma severe weather [1].

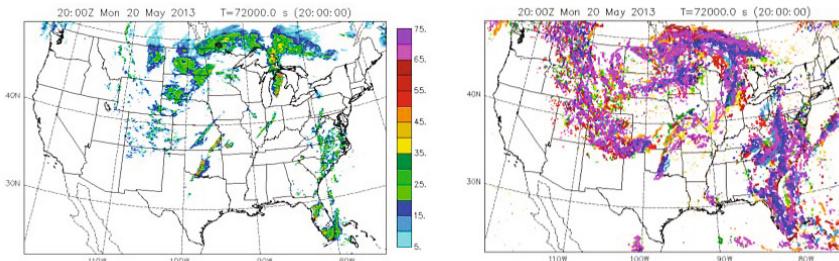


Fig. 1. 20 hour forecast initiated from 00 UTC May 20, 2013, valid at 20 UTC May 20 at the time of the Moore tornado touch down. Left: radar observation; Right: ensemble spaghetti chart of forecast radar reflectivity exceeding 35 dBZ.

3.3 Light-Water Reactor Modeling

The Consortium for Advanced Simulation of Light Water Reactors (CASL) was established as the first U.S. Department of Energy (DOE) Innovation Hub, and was created to accelerate the application of advanced modeling and simulation (M&S) of nuclear reactors. CASL applies existing M&S capabilities and develops

advanced capabilities to create a usable environment for predictive simulation of light water reactors (LWRs). This environment, known as the Virtual Environment for Reactor Applications (VERA), incorporates science-based models, state-of-the-art numerical methods, modern computational science and engineering practices, and uncertainty quantification (UQ) and validation against data from operating pressurized water reactors, single-effect experiments, and integral tests.

The heaviest usage of Darter was as a development platform for a physics component known as MPACT, being developed primarily by staff and students at the University of Michigan. MPACT is one of two components being developed to simulate the behavior of neutrons within reactor cores. In addition to development activities, MPACT performed multiple simulations that used over 3,000 cores for roughly 3.5 hours. A result of one of these simulations is depicted in Fig. 2. It should also be noted that the value of Darter to CASL went beyond merely the delivery of computational cycles. First, it provided a different software environment for VERA, and each additional platform provides an opportunity improve VERA's portability. Second, Darter provided a new and unique hardware configuration for VERA, being the first Intel-based Cray system CASL had used, as well as the Aries interconnect technology.

One of the most significant milestones CASL has delivered, simulation of the Watts Bar Unit 1 zero power physics test, was completed successfully in early June of 2013 [14]. Successful completion of this milestone involved multiple teams within CASL and a variety of computational resources. The Darter system played a significant role, by providing development and analysis cycles at a critical time.

3.4 Nucleosynthesis in Modeling of Core-Collapse Supernovae

Using the Darter system, the UTK-ORNL Astrophysics Theory Group (UOAstro) is taking advantage of recent technological improvements in scalability and performance in high-performance computing. UOAstro uses their multidimensional radiation hydrodynamics code, CHIMERA [15], to make substantial strides in furthering our understanding of the explosion mechanism in core-collapse

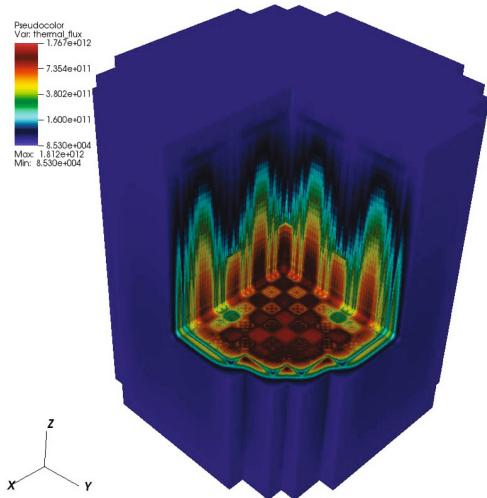


Fig. 2. Thermal Neutron Flux Distribution Generated by MPACT on Darter

supernovae. CHIMERA has been used to do supernovae modeling on previous generations of Cray machines and has easily been ported to run on Darter. On Darter, the team coupled their thermonuclear reaction network code, XNet [16], to CHIMERA to evolve the nucleosynthesis of a 150-species nuclear network.

Core-collapse supernovae are the violent death of massive stars and are among the most powerful explosions in the universe. These explosions last for only few tens of seconds while releasing about 10^{53} erg of energy, rivaling the instantaneous power of all the rest of the luminous visible universe combined. During their lifetime, massive stars are the dominant sites where elements in the periodic table between oxygen and iron are synthesized in the process known as *stellar nucleosynthesis*.

Nucleosynthesis does not cease during the explosion event of core-collapse supernovae. In fact, there is a growing evidence that core-collapse supernova are responsible for producing half the elements heavier than iron [17] in a process known as *r-process nucleosynthesis*. Supernovae nucleosynthesis also produces and deposits energy to the supernova ejecta, and the details of this process may be important ingredients of the explosion mechanism. Yet, a frequently employed computational economy in the *ab initio* core-collapse supernova modeling has been the simplification of the nucleosynthesis network to at most a 14-species α -network. These simplified networks fail to accurately describe both the composition and energy distribution of supernovae ejecta as directly observed [18].

At present, UOAsstro is evolving a two-dimensional core-collapse supernova model from the inner core of a 15 solar mass, non-rotating progenitor [19] on a polar axisymmetric grid with 720 radial zones and 240 angular zones and a 150-species nuclear reaction network. Early results are promising, and show quantitative differences with simulations from the same model with an α -network.

Qualitative comparisons, however, must be postponed until the explosion energy asymptotes and tracer particle fates are determined, marking the “completion” of the model. Prior experience has shown that this is typically achieved after roughly 1.5 seconds of evolution. After 10 days of computation on 1,920 cores on Darter, the model has evolved 650 ms in total, or 320 ms since the end of stellar

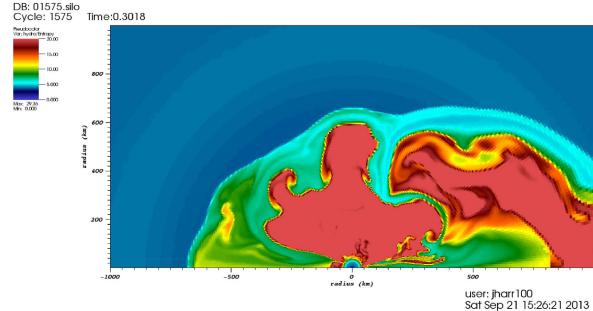


Fig. 3. Entropy plot of the model with 150-species nuclear network at 300 ms after core bounce, where some of the characteristic behavior of the explosion, such as SASI [20] is apparent

collapse, and the average shock radius has reached approximately 880 km, as seen in Fig. 3.

4 Conclusions

UTK acquired a Cray XC30, called Darter, with a peak performance of 248.9 Teraflops. Darter was deployed in late March of 2013 with a very aggressive timeline to be in production: the system was deployed, accepted, and placed into production in only 2 weeks. The Spring Experiment for the Center for Analysis and Prediction of Storms largely drove the accelerated timeline, as the experiment was scheduled to start in mid-April. Scientists from the CAPS immediately began functionality testing as they worked to port their scientific models to this new computing architecture. The CASL project also needed access and was able to meet their tight deadlines on the newly acquired XC30. The rapid deployment and transition to operations could not have been possible without the strong partnership with Cray. Application teams have benefited highly from the consistency of the Cray multigenerational software stack. This has truly been a successful deployment.

Acknowledgments. We would like to thank Ming Xue for contribution of the CAPS highlight, John Turner for the contribution of the CASL highlight, and James Austin for the contribution of the UTK-ORNL Astrophysics highlight. We need to acknowledge the help we received from the National Center for Computational Sciences at ORNL during the acceptance phase, and in particular we would like to thank Buddy Bland, Matt Ezell, and Arnold Tharrington. We would also like to thank Gary Rogers for putting together the Darter system statistics.

This material is based upon work performed using computational resources supported by the University of Tennessee and Oak Ridge National Laboratory Joint Institute for Computational Sciences.

References

1. Woodie, A.: Cray Supercomputer Gave Forecasters an Edge in Tornado Prediction. HPCWire (August 2013),
http://archive.hpcwire.com/hpcwire/2013-08-12/cray_supercomputer_gave_forecasters_an_edge_in_tornado_prediction.html
2. Marr, D.T., Binns, F., Hill, D.L., Hinton, G., Koufaty, D.A., Miller, J.A., Upton, M.: Hyper-threading technology architecture and microarchitecture. Intel Technology Journal 6 (2002)
3. Zhao, Z., Wright, N.J., Antypas, K.: Effects of Hyper-Threading on the NERSC workload on Edison Lawrence Berkeley National Laboratory. In: [21]
4. Leng, T., Ali, R., Hsieh, J., Stanton, C.: A Study of Hyper-Threading in High-Performance Computing Clusters. Dell 6(1), 33–36 (2002)

5. Faanes, G., Bataineh, A., Roweth, D., Court, T., Froese, E., Alverson, B., Johnson, T., Kopnick, J., Higgins, M., Reinhard, J.: Cray Cascade: A scalable HPC system based on a Dragonfly network. In: Proceedings of the 2012 International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2012, pp. 1–9. IEEE Computer Society, Washington, DC (2012)
6. Oracle: Lustre 2.0 Operations Manual. Oracle (2011)
7. Padua, D. (ed.): Encyclopedia of Parallel Computing. Springer (2011)
8. Fahey, M., Jones, N., Hadri, B.: The automatic library tracking database (ALTD). In: Proceedings of the 2010 CUG Conference, Edinburgh, Scotland (May 2010)
9. Petitet, A., Whaley, R.C., Dongarra, J.J., Cleary, A.: A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. Innovative Computing Laboratory (September 2000), <http://icl.cs.utk.edu/hpl/>
10. Laboratory, I.C.: HPC Challenge Benchmarks, <http://icl.cs.utk.edu/hpcc/>
11. of the University of California, T.R.: IOR., <https://github.com/chaos/ior>
12. for Atmospheric Research, U.C.: The Weather Research and Forecasting Model, <http://www.wrf\discretionary{-}{-}{-}model.org/index.php>
13. You, H., Budiardja, R., Betro, V., Hadri, B., Patel, P., Logan, J., Fahey, M., Crosby, L.: Performance comparison of scientific applications on Cray architectures. In: [21]
14. Gehin, J., Godfrey, A., Franceschini, F., Evans, T., Collins, B., Hamilton, S.: Operational reactor model demonstration with VERA: Watts bar unit 1 cycle 1 zero power physics tests. CASL L1 Milestone Report CASL.L1.P7.01, CASL-U-2013-0105-000 (June 2013)
15. Bruenn, S.W., Mezzacappa, A., Hix, W.R., Blondin, J.M., Marronetti, P., Messer, O.E.B., Dirk, C.J., Yoshida, S.: 2D and 3D core-collapse supernovae simulation results obtained with the CHIMERA code. Journal of Physics: Conference Series 180, 012018 (2009)
16. Hix, W.R., Meyer, B.S.: Thermonuclear kinetics in astrophysics. Nuclear Physics A 777, 188–207 (2006)
17. Argast, D., Samland, M., Thielemann, F.K., Qian, Y.Z.: Neutron star mergers versus core-collapse supernovae as dominant r-process sites in the early Galaxy. Astronomy and Astrophysics 416(3), 997–1011 (2004)
18. Timmes, F.X., Hoffman, R.D., Woosley, S.: An Inexpensive Nuclear Energy Generation Network for Stellar Hydrodynamics. The Astrophysical Journal Supplement Series 129(1), 377–398 (2000)
19. Woosley, S., Heger, A.: Nucleosynthesis and remnants in massive stars of solar metallicity. Physics Reports 442(1-6), 269–283 (2007)
20. Blondin, J.M., Mezzacappa, A., DeMarino, C.: Stability of Standing Accretion Shocks, with an Eye toward Core-Collapse Supernovae. The Astrophysical Journal 584(2), 971–980 (2003)
21. The Cray User Group, Inc.: Proceedings of the 2013 CUG Conference. In: Proceedings of the 2013 CUG Conference, Napa, CA (May 2013)

Cyme: A Library Maximizing SIMD Computation on User-Defined Containers

Timothée Ewart, Fabien Delalondre, and Felix Schürmann

Blue Brain Project, Brain Mind Institute, EPFL, Switzerland

timothee.ewart@epfl.ch

Abstract. This paper presents Cyme, a C++ library aiming at abstracting the usage of SIMD instructions while maximizing the usage of the underlying hardware. Unlike similar efforts such as Boost SIMD or VC, Cyme provides generic high level containers to the users which hides SIMD complexity. Cyme accomplishes this by 1) optimization of the Abstract Syntax Tree using Expression Templates Programming to prevent temporary copies and maximize the use of *Fuse Multiply Add* instructions and 2) creating a data layout in memory (AoS or AoSoA), which minimizes data addressing and manipulation throughout all SIMD registers. Implementation of Cyme library has been accomplished on the IBM Blue Gene/Q architecture using the 256 bit SIMD extensions (QPX) of the Power A2 processor. Functionality of the library is demonstrated on a computationally intensive kernel of a neuro-scientific application where an increase of GFlop/s performance by a factor of 6.72 over the original implementation is observed using Clang compiler.

Keywords: SIMD, Vectorization, Memory layout, C++, Generic Programming.

1 Introduction

Most CPUs available on today's market implement instruction sets (SSE, AVX (x86) and Altivec (power)) which allow processing on multiple (vectorized) data sets, known as Single Instruction Multiple Data (SIMD) [1]. Considering the costs involved, optimizing scientific applications to be executed on massively parallel machines made of millions of cores requires making maximum usage of available hardware utilization using SIMD programming. As such a task requires expert hardware design and low level programming knowledge most of the scientific applications either rely on compiler technology and autovectorization to introduce SIMD or programmers to implement most computationally intensive kernels using compiler supported `#pragma` directives.

Those methods provide maximum performance for kernels where a clear vectorizable pattern can be identified such as applications relying on the resolution of very large linear algebraic systems. However, when the code includes uncertainties due to complex data dependencies, non-contiguous memory access, aliasing or uncountable loops, compilers either generate poorly vectorized or scalar

code [1]. In cases where leveraging from compiler technology is not applicable, High Performance Computing (HPC) expert knowledge is usually required to implement kernels using intrinsics and methods favoring autovectorization. This software development model is efficient and sustainable for high-performance applications that depend on few critical kernels and want to reach a high level of performance suitable for competing for the Gordon Bell Prize.

When it is necessary to implement thousands of critical kernels on many platforms and autovectorization is difficult, the development models discussed previously become unsustainable. To better support such a use case, new libraries [2,3] have been developed which provide a higher level of abstraction via an *Embedded Domain Specific Language*.

VC [3] library relies on the definition of a vector of size equal to the architecture register length. Using such a vector, developers implement critical kernels by defining operations on the VC vector rather than the data structure defined by scientists. In addition, VC provides a memory container to ease the use of arrays which packs VC vectors together to support **Structure of Array (SoA)** internal memory layout. Such a container allows automatically aligning and padding the memory to allow fast vector access in the full index range. Additionally, it provides member functions to easily iterate over all vectors or scalars in the array. If this approach proves to provide good performance on x86 (SSE4-AVX) architecture, it forces the developer to either perform critical operations on data structure that differs from the scientifically defined one or make use of arrays.

A similar level of abstraction is supported by Boostsimd [2] which relies on STL containers and the definition of transformation functions (functors). The bulk of the work implied by this method consists of implementing the necessary functors for every transformation and architecture. With the data properly laid out in memory, Boostsimd relies on using Expression Templates [4] to construct and optimize the Abstract Syntax Tree (AST) at compile-time. While this solution provides an elegant framework publicly accessible on x86 (SSE4-AVX) architecture to support data structures more suitable to scientific applications such as **Array of Structure (AoS)**, it still requires highly skilled developers.

A few years ago Intel developed the Array Building Blocks (ArBB) library which relied on generative metaprogramming and just-in-time compilation to generate optimized code on Intel hardware. If this approach was interesting, the fact that it is no longer supported prevents from performing a fair comparison with VC and Boostsimd. Nevertheless, it is worthwhile to point out that contrary to other available solutions ArBB was only available on Intel platforms which limited its adoption on available supercomputing architectures [5].

In this paper, a high level C++ SIMD library called Cyme¹ which primarily aims at optimizing a class of applications that relies on the resolution of a large number of small but similar kernels is presented. It provides a higher level of abstraction compared to available solutions [2,3] which is believed to be

¹ Cyme (noun botany) a flower cluster with a central stem bearing a single terminal flower that develops first, and other flowers in the cluster developing as terminal buds of lateral stems. (Good analogy to the Abstract Syntax Tree in Computer Science.)

more suitable to scientific developer while maximizing usage of the underlying hardware. Using such a library the scientist can continue developing generic algorithms that make use of rather complex data structures made of the same variable type independently of the actual layout of the data in memory (**AoS** and **Array of Structure of Array (AoSoA)**).

The paper is then organized as follows: In the second part of the paper, Cyme library design and implementation are introduced. Implementation details are presented as well as descriptions of both user and internal workflows. The third section details the integration of Cyme in a life science micro-kernel. Results are first compared against a reference solution on a simple example before a complete performance analysis that includes performance modeling is provided.

2 Cyme Design and Implementation

Cyme overall design is presented in Figure 1. The library is made of three main components: containers and operators which are made accessible to Cyme user and a vector component which is internal to Cyme workflow and implementation.

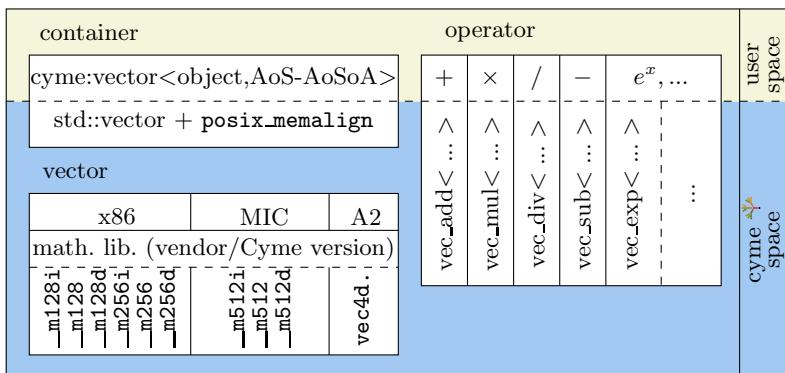


Fig. 1. Cyme components and overall design

Containers. are used to define large (macro) data sets where every item of the macro set is a rather complex but identical (micro) structure or object containing different values of the same type. To maximize performance Cyme library provides two macro containers derived from STL: `cyme::array` and `cyme::vector`. From a user point of view, the declaration of a macro Cyme container is completely generic and only requires:

1) Defining a micro structure which contains the type as well as the number of variables to be stored. In the example provided below, the user defines the micro container such that it contains 5 double precision numbers.

```
1 struct object{  
2     typedef double value_type;  
3     const static int value_size = 5;}
```

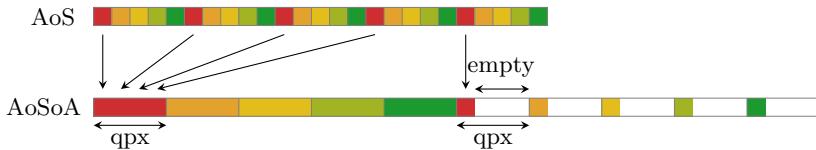


Fig. 2. `cyme::array` of 5 elements of type `object` laid out in memory according to AoS (top) and AoSoA (bottom) pattern. When using AoSoA layout the data is packed over the size of the native QPX register e.g. 4 `double` precision numbers.

2) Defining a macro structure or Cyme container by specifying the micro structure to be used as well as the internal memory layout. In the example provided below, the user defines a Cyme array called "block" of size 5 (i.e. 5 micro structures made of 5 double precision numbers) laid out in memory using AoSoA.

```
1   cyme::array<object,5,AoSoA> block;
```

Taking this information into account, Cyme containers allocate data either in DRAM memory (`cyme::vector`) or on the stack (`cyme::array`) and packs all microstructures according to the prescribed memory layout (AoS or AoSoA). Such a packing includes memory interleaving where the interleaved block sizes are of the size of the native SIMD register. To illustrate this operation, consider an example where the microstructure contains 5 elements and represented by the following scheme: , one can alternatively declare an array using either `cyme::array<object,5,AoSoA>` or `cyme::array<object,5,AoS>`, (Figure 2). With such a generic interface and the addition of appropriate iterators, it becomes easy for a Cyme user to traverse and manipulate data (Figure 3) following his conceptual representation while a SIMD (AoSoA) optimized memory layout is in fact implemented.

Operators. map mathematical expressions implemented in the user space to the SIMD vector component. All operations are determined at compile time and can therefore be encoded in template operations. When an operator is called, it first returns an intermediate object specific to the corresponding operation (`vec_add`, `vec_mul`, ...). The later is then reused as template parameter in subsequent operations, building a first version of an optimized AST. At completion, the assignment operator is called and computation of each node of the tree is done recursively. During this phase Cyme operators are replaced by wrappers supported by the Cyme vectors. Such a recursive mechanism allows optimizing the AST tree by minimizing the use of temporary copies but also by introducing complex instructions such as Fused Multiply Add (FMA). The latter is for example supported by defining several template signatures of the addition operator associated with the multiply one (`vec_mul`). When a `vec_mul` is created, if the following recursive operation is an addition, it will be caught by a specific addition operator which is associated with a FMA.

```

1   cyme::array<object,AoSoA or AoS> block;
2   typename::array<object,AoSoA or AoS>::iterator it ;
3   for( it = block.begin(); it != block.end(); ++it)
4       (*it )[0] = (*it )[1]x(*it )[2]+(*it )[3];

```

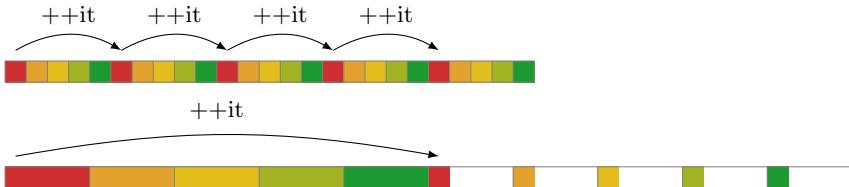


Fig. 3. Abstraction of the internal memory layout and generic algorithm implementation: Using iterators and Cyme containers, users implement the very same code while different operations are performed. AoSoA layout (bottom) allows performing 4 operations ($QPX = 4 \text{ double}$) in a single SIMD instruction while AoS (top) executes sequential operations.

Vectors. abstract SIMD register/intrinsics and external library using encapsulation. Its implementation is via a trait class which allows providing architecture specific implementations. Cyme vectors are being used during the last phase of the AST construction where the AST nodes that are first supported by Cyme operators are replaced by SIMD intrinsics or external calls to mathematical libraries. The later can either be system/vendor or user defined libraries. It is worthwhile to point out that both the e^x (Cephes) and division (using Newton-Raphson method) operations have been re-implemented to increase performance.

Although VC and Boostsimd librairies have not been fully interfaced with Cyme yet, being able to plug them to replace part of the Cyme components would add considerable value and versatility. Preliminary experiments suggest that VC could replace Cyme component vector while Boostsimd could replace both operator and vector components.

3 Application to Life Science Micro-kernels

The goal of the Blue Brain Project [6] is to unify our understanding about the rodent brain with the help of detailed computer models. The detailed cellular level brain tissue models [7] are simulated using the open source simulator NEURON [8]. The NEURON application supports modeling the electrical activity of neuronal networks using the cable theory [9]. Every neuron is here represented with a detailed morphology discretized into hundreds of electrical compartments [7]. Some simulations of the Blue Brain Project include about 2-3 channel types per compartment and 8,000 synapses per neuron where the definition of each channel or synapse model instance (one channel or synapse) requires storing from 15 to 30 floating point variables. Since brains have a large amount of intrinsic parallelism due to the large number of neurons (e.g. an estimated 200 million neurons and 10^{12} synapses for an entire rat brain [10]), synapses and channels

```

void rates(double *p){
    if(v== -35){v=v+0.0001;}
    mAlpha=(0.182*(v+35))/(1.0-(exp(-(v+35)/9)));
    mBeta=(0.124*(-v-35))/(1.0-(exp((v+35)/9)));
    mInf=mAlpha/(mAlpha+mBeta) ;
    mTau=1/(mAlpha+mBeta) ;
    if (v== -50) {v=v+0.0001;}
    hAlpha=(0.024*(v+50))/(1-(exp(-(v+50)/5)));
    if (v== -75) {v=v+0.0001;}
    hBeta=(0.0091*(-v-75))/(1-(exp(-(-v-75)/5)));
    hInf=1.0/(1.0+exp((-v-65.0)/6.2));
    hTau=1.0/(hAlpha+hBeta);
}

void states (double* p){
    rates(p);
    m+=(1-exp(-dt/mTau))
        *(- (mInf/mTau)/(-1/mTau)-m);
    h+=(1-exp(-dt/hTau))
        *(- (hInf/hTau)/(-1/hTau)-h);
}

```

```

template<class T> static inline
void rates(typename T::storage_type& S){
    S[8]=(0.182*(S[16]+35))
        /(1-(exp((-35-S[16])/9)));
    S[9]=(-0.124*(S[16]+35))
        /(1-(exp((S[16]+35)/9)));
    S[6]=S[8]/(S[8]+S[9]);
    S[7]=1./(S[8]+S[9]);
    S[12]=(0.024*(S[16]+50))
        /(1-(exp((-50-S[16])/5)));
    S[13]=(-0.0091*(S[16]+75))
        /(1-(exp((S[16]+75)/5)));
    S[10]=1/(1+exp((S[16]+65)/6.2));
    S[11]=1/(S[12]+S[13]);
}

template<class T> static inline
void states(typename T::storage_type& S){
    rates<T>(S);
    S[3]+=(1-exp(dt/S[7]))*(S[6] -S[3]);
    S[4]+=(1-exp(-dt/S[11]))*(-(S[10]-S[4]));
}

```

Fig. 4. Implementation of Na Channel Model from the original and cyme version

of the same type will be instantiated a large number of times per node, making them good candidates for SIMD programming.

3.1 Blue Brain Mini-application Library: Performance Analysis and Modeling of Na Channel Model Implementation

To be able to easily test new features and optimizations, the Blue Brain Project is implementing a mini-application library which aims at reproducing the main features of the simulation software. For the sake of this paper, a sodium (Na) channel model has been extracted from the simulation software and three versions of this models have been implemented using Cyme (Figure 4): 1) The original version found in NEURON software and two versions implemented using 2) Cyme AoS and 3) Cyme AoSoA containers.

Comparison with Boost.simd on x86 Architecture: The Cyme library has primarily been developed and implemented to support applications that include a large number of small but similar kernels running on IBM Blue Gene/Q. Nevertheless, to increase portability and offer solutions on multiple major hardware infrastructures, porting of the library on x86 architecture with SSE/AVX technologies has been carried out. Such a porting allows evaluating SIMD code optimization quality generated when using Cyme by comparing it with a reference solution (Boostsimd). As an example, the assembly code generated by Clang compiler (V3.2) when using both frameworks is presented in Table 1 for a simple expression ($S[0] = S[4](S[5] + S[6]) + S[1]S[2]$). In both cases it can be observed that the number of total instructions is the same and `mov` instructions are reduced to a minimum, validating Cyme library implementation.

Performance Analysis and Modeling on IBM Blue Gene/Q: The A2 processor of the IBM Blue Gene/Q [11] is a native 16 core at 1.6 GHz which delivers a peak performance of 204.8 Gflop/s. Each A2 core is capable four-way

Table 1. Assembly code generated by Clang V3.2 compiler when using Boost.simd (left) or cyme (right) libraries on an Intel i5-2520M machine

	boost.simd		cyme
vmovapd	-128(rsp,rax),yymm0	vmovapd	-64(rsp,rax),yymm0
vmovapd	(rsp,rax),yymm1	vmovapd	64(rsp,rax),yymm1
vmulpd	-96(rsp,rax),yymm0,yymm0	vmulpd	-32(rsp,rax),yymm0,yymm0
vaddd	32(rsp,rax),yymm1,yymm1	vaddd	96(rsp,rax),yymm1,yymm1
vmulpd	-32(rsp,rax),yymm1,yymm1	vmulpd	32(rsp,rax),yymm1,yymm1
vaddd	yymm1, yymm0,yymm0	vaddd	yymm1, yymm0,yymm0
vmovapd	yymm0, -160(rsp,rax)	vmovapd	yymm0, -96(rsp,rax)

multithreading and has 16k+16k data cache. All cores execute instructions in order, and provide 32 QPX 256-bit registers. Each of the 32 registers can contain four elements of 64 bits, where each of the execution slot operates on one vector element. A2 processor supports complex instructions such as FMA operation.

As Cyme implementation heavily relies on the use of highly templated C++ code its performance depends upon the ability of the compiler to efficiently support C++ expression templates. Three compilers are available on Blue Gene/Q: **IBM XLC (V12.1):** Officially shipped by IBM, XLC supports QPX intrinsics and both OpenMP/Pthreads. It is traditionally used by HPC applications to get maximum usage of the QPX floating point unit. The latest version does however not efficiently support C++ Expression Templates nor C++11 standard. As a consequence, highly templated codes deliver poor performance.

GNU GCC (V4.4.7): Added to Blue Gene/Q for convenience, GCC does not support QPX intrinsics which prevents codes heavily relying on a large number of floating point instructions to efficiently execute on Blue Gene/Q.

Clang (V3.4): Supports both highly templated code and C++11 standard. Thanks to Hal Finkel [12], both QPX and more recently OpenMP threading model supports have been added on Blue Gene/Q, making it an extremely good candidate to help Cyme reaching maximum performance.

Performance comparison of the three implementations of the Na channel model (Figure 4) are depicted in Figures 56 and Table 2. All floating point and operational intensity numbers have been obtained using HPM library [13]. From Figure 5, it can first be observed that both the original and AoS implementations converge to the same level of performance when solving a large number (greater than 1,000) of Na channel models. This confirms that both implementations deliver the same performance for configurations used in production by the Blue Brain Project. It is worth mentioning that the observed difference in GFlop/s performance obtained for a lower number of Na channels is unknown at this point in time and is being investigated. More importantly, one can clearly observe the large difference of performance when using AoSoA layout and Clang compiler compared to all the other implementations for a large number of channel models, confirming the usefulness of the method described in this article. It is also worth noting that, due to the lack of support for C++ expression template, the XLC compiler was not able to take advantage of the AoSoA memory layout and delivered lower performance than both the original and AoS implementations.

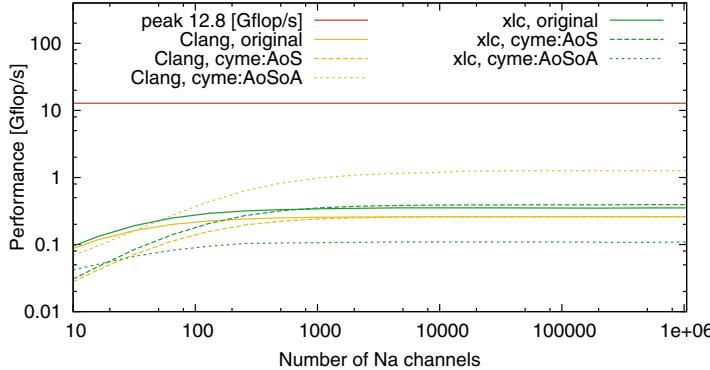


Fig. 5. GFlop/s performance for the original and Cyme AoS/AoSoA implementations of the Na channel model using a single thread of the IBM Blue Gene/Q A2 processor. Results are provided for both Clang and XLC compilers.

Table 2. GFlop/s performance, operational intensity and time to solution when solving 1.0×10^6 Na channel using the original, AoS and AoSoA implementations using the 64 threads of IBM Blue Gene/Q A2 processor

	Perf. [Gflop/s]		Op. In. [flop/Byte]		Time [s]	
	Clang	xlc	Clang	xlc	Clang	xlc
original	6.20	19.67	1.14	1.37	0.066	-1.020
Cyme:AoS	14.23	11.38	2.12	1.80	0.030	0.027
Cyme:AoSoA	41.71	2.09	1.58	5.05	0.010	0.36
Best peak fraction [%]	20.36%	9.60%	-	-	-	-
Improvement	6.72×	-	1.39×	-	6.66×	-

The results of table 2 confirm those obtained in Figure 5 for a single thread. The implementation using AoSoA and Clang compiler was able to reach 20.36% of the A2 processor peak performance which consists of a factor of 6.72 increase compared to the original version. However the computational intensity only increased by 39% compared to the original version. The full analysis of such a result is not yet complete but to date implementations of both Cyme and Clang OpenMP support on Blue Gene/Q are known to not yet be optimal in terms of data movement. For example, to date implementation of the Na channel model using Cyme brings non used data from DRAM to register which decreases the maximum operational intensity. Although not yet ready to be released at the time of submission of this article, preliminarily results show potential increase of performance by about 20-30%. In addition, at the time of the edition of this article support of OpenMP on Blue Gene/Q by Clang compiler is still experimental and may not yet take full advantage of available prefetching strategy.

By extracting numbers from Table 2, a roofline [14] model is constructed and presented in Figure 6. From this graph, one can conclude that the current implementation of the Na channel model using AoSoA and Clang maximizes the GFlop/s performance on BG/Q A2 processor for the given operational intensity.

$$\text{Attainable [Gflop/s]} = \min \begin{cases} \text{Peak performance [Gflop/s]} \\ \text{Peak Mem. Bandwidth} \times \text{Op. Intensity} \end{cases} \quad (1)$$

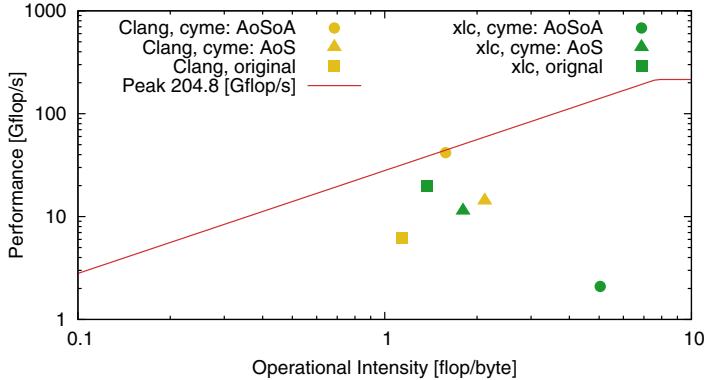


Fig. 6. Roofline [14] model representing the performance of the original and AoS/AoSoA Cyme implementations when using 64 threads of the IBM Blue Gene/Q A2 processor. Results are provided for both Clang and XLC compilers. The Operational Intensity is calculated by the ratio Gflop/Bandwidth where the bandwidth is the memory transfert between DRAM and cache (HPM library).

4 Conclusions and Future Work

A high level C++ library aiming at abstracting SIMD programming has been presented. Contrary to similar high level libraries [2,3], Cyme provides user data containers which completely hide SIMD programming complexity from the user standpoint. This allows scientists to continue developing software with data structures that are more natural to conceptualize while the actual memory layout (AoSoA) is designed to maximize the usage of SIMD instructions. Using those containers and the method of Expression Templates which supports optimizing the Abstract Syntax Tree, a first simple expression was validated against Boost SIMD by comparing the assembly code produced by both libraries. The analysis of Cyme performance was then carried out on a life science use case (sodium channel) where many small similar kernels are being resolved on IBM Blue Gene/Q A2 processor. Such an analysis showed that the AoSoA memory layout was able to considerably improve performance (factor 6.72) to reach up to 20.36% of the A2 processor peak performance using the Clang compiler. To understand whether it was possible to reach even higher peak performance, a roofline model was constructed. The latter showed that the maximum peak performance was obtained for the given operational intensity (1.58).

Recent experiments not included in this paper showed that the Na channel model implementation can be further improved by more carefully selecting the data to be brought from DRAM memory to registers. As a consequence, immediate Cyme internal developments will focus on the latter as well as investigating the possibility to take full advantage of prefetching technologies. In

addition, future long term developments will likely include fully interfacing with VC and Boostsimd libraries as well as porting Cyme to other architectures.

On the user application development side, the full integration of Cyme in simulation software developed by the Blue Brain Project to implement both channel and synapse models will be carried out. Although computational neuroscience software performance and scalability depend on a number of other factors, the performance analysis of such software showed that this will provide considerable speed up to the class of applications which include a large number of small but similar kernels. As this work was inspired by such a type of use cases, application of Cyme to new problems will also be part of future work.

Acknowledgments. The authors of the present paper would like to greatly thank the members of the Blue Brain Project HPC team, the Swiss Center for Scientific Computing, Hal Finkel and Bob Walkup for the many fruitfull discussions and feedback they provided. The EPFL Blue Brain Project, the Blue Brain IV BlueGene/Q system as well as this study are funded by the ETH board. The financial support for CADMOS and the Lemanicus Blue Gene/Q system is provided by the Canton of Geneva, Canton of Vaud, Hans Wilsdorf Foundation, Louis-Jeantet Foundation, University of Geneva, University of Lausanne, and Ecole Polytechnique Fédérale de Lausanne.

References

1. Bik, A.J.C.: Software Vectorization Handbook, The: Applying Intel Multimedia Extensions for Maximum Performance. Intel Press (2004)
2. Esterie, P., Gaunard, M., Falcou, J., Lapresté, J.T., Rozoy, B.: Boostsimd: generic programming for portable SIMDization. In: PACT, pp. 431–432. ACM (2012)
3. Kretz, M., Lindenstruth, V.: Vc: A c++ library for explicit vectorization. Software: Practice and Experience 42(11), 1409–1430 (2012)
4. Vandevord, D., Josuttis, N.M.: C++ Templates. Addison-Wesley (2002)
5. <http://software.intel.com/en-us/articles/intel-array-building-blocks>
6. Markram, H.: The blue brain project. Nature reviews. Neuroscience 7(2) (2006)
7. Hay, E., Hill, S., Schürmann, F., Markram, H., Segev, I.: Models of neocortical layer 5b pyramidal cells capturing a wide range of dendritic and perisomatic active properties. PLoS Comput. Biol. 7(7) (2011)
8. <http://www.neuron.yale.edu/neuron/>
9. Core Conductor Theory and Cable Properties of Neurons. J. Wiley & Sons (2011)
10. Herculano-Houzel, S., Mota, B., Lent, R.: Cellular scaling rules for rodent brains. Proceedings of the National Academy of Sciences of the United States of America 103(32), 12138–12143 (2006)
11. IBM System Blue Gene Solution: BG/Q Application Development. IBM (2013)
12. Finkel, H.: <http://trac.alcf.anl.gov/projects/llvm-bgq>
13. Salapura, V., Ganeshan, K., Gara, A., Gschwind, M., Sexton, J.C., Walkup, R.: Next-generation performance counters: Towards monitoring over thousand concurrent events. In: ISPASS, pp. 139–146. IEEE (2008)
14. Williams, S., Waterman, A., Patterson, D.: Roofline: An insightful visual performance model for multicore architectures. Commun. ACM 52(4), 65–76 (2009)

A Compiler-Assisted OpenMP Migration Method Based on Automatic Parallelizing Information

Kazuhiko Komatsu^{1,3}, Ryusuke Egawa^{1,3},
Hiroyuki Takizawa^{2,3}, and Hiroaki Kobayashi¹

¹ Cyberscience Center, Tohoku University,
6-3 Aoba, Aramaki-aza, Aobaku, Sendai, 980-8578, Japan
`{komatsu,egawa,tacky,koba}@isc.tohoku.ac.jp`
`http://www.sc.isc.tohoku.ac.jp/`

² Graduate School of Information Sciences, Tohoku University,
6-3 Aoba, Aramaki-aza, Aobaku, Sendai, 980-8578, Japan

³ Core Research for Evolutional Science and Technology,
Japan Science and Technology Agency (JST CREST)

Abstract. Performance of a serial code often relies on compilers' capabilities for automatic parallelization. In such a case, the performance is not portable to a new system because a new compiler on the new system may be unable to effectively parallelize the code originally developed assuming a particular target compiler. As the compiler messages from the target compiler are still useful to identify key kernels that should be optimized even for the different system, this paper proposes a method to migrate a serial code to the OpenMP programming model by using such compiler messages. The aim of the proposed method is to improve the performance portability across different systems and compilers. Experimental results indicate that the migrated OpenMP code can achieve a comparable or even better performance than the original code with automatic parallelization.

Keywords: OpenMP migration, Performance portability, Automatic parallelization.

1 Introduction

The number of cores on recent HPC systems has been drastically increased to enhance the computational capability. In order to exploit such computational capabilities of many cores, a code has to be parallelized so as to use multiple threads, called multithread parallelization. One of the most widely-used approaches to multithread parallelization is to use automatic parallelization functions of compilers. Most of recent compilers, especially commercial compilers, provide their own automatic parallelization functions. One of the most beneficial advantages is the productivity. All a user has to do is to compile a code with automatic optimization options. Since no code modifications are necessary for automatic

optimizations, the user can easily try and enjoy the computational capability of many cores.

Another advantage is that a compiler can generate efficient object codes[1]. By considering the characteristics of a target system, only appropriate parallelization for the target system can selectively be employed. Furthermore, due to the easiness to use automatic parallelization, most of codes are modified by hand so that a particular compiler can easily understand the code structure, and analyze the data dependency. Hereafter, the compiler assumed by an original code is called the target compiler. As a result, the performance of an automatically parallelized code can be higher than that of a manually parallelized one by the user.

However, as automatic optimizations of a compiler are carried out to exploit the potentials of a target system, the performances of codes are not always portable to other systems with different compilers. As many types of recent HPC systems have been developed, the performance portability across different types of HPC systems becomes more important. To improve the performance portability of a code, OpenMP parallelization is one of the promising candidates. Although appropriate OpenMP directives need to explicitly be inserted to parallel regions of a code, OpenMP codes can be executed on any HPC system. Accordingly, the performance portability is expected to improve by migrating a serial code to the OpenMP model, called OpenMP migration.

This paper proposes a compiler-assisted OpenMP migration method that can easily adapt an application code to OpenMP compilers. In conventional OpenMP migration, it is labor-intensive even to find parallelizable loop nests in a large-scale HPC application. The key idea of the proposed method is to use the compiler messages from the target compiler of the application. Those messages could be informative to improve the performance portability because clues for the OpenMP parallelization obtained from the messages are often effective for different systems as well as the target system. Therefore, the proposed OpenMP migration method utilizes the compiler messages to help a user easily identify parallelizable loop nests.

The rest of this paper is organized as follows. Section 2 proposes a compiler-assisted OpenMP migration method to improve the performance portability at a low programming effort. Section 3 evaluates the performances of the migrated OpenMP codes on various HPC systems. Section 4 briefly describes the related work about compiler-assisted OpenMP migration. Finally, Section 5 gives conclusions of this paper.

2 A Compiler-Assisted OpenMP Migration Method

This paper proposes a compiler-assisted OpenMP migration method that helps a programmer find parallelizable loop nests and insert appropriate OpenMP directives. The key idea is to utilize the messages from a compiler, for which an HPC code has been performed automatic optimizations. In general, an HPC code is optimized assuming a particular target compiler. As a result, the target compiler

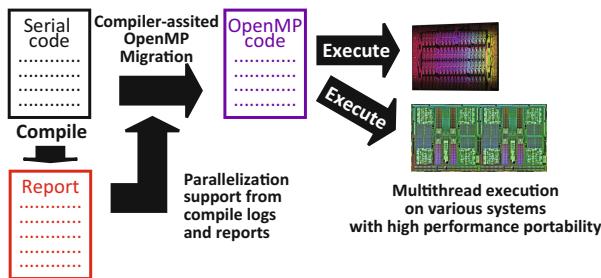


Fig. 1. Compiler-assisted OpenMP migration

can detect the parallelism and data dependencies necessary for multithread parallelization. Therefore, the information about how the target compiler performs automatic optimizations is helpful to guess programmer's intention for the optimization. Figure 1 shows the overview of the proposed migration method. By making full use of compiler messages from the target compiler, the method can facilitate the migration of a serial code to the OpenMP model.

A compiler generally produces compiler messages that show detailed information about automatic optimizations such as parallelization, vectorizations, and loop transformations. The compiler messages describe which part in a source code is parallelized and vectorized, which part is treated as a critical section, what optimization is applied, and so on. Listing 1.1 shows an example of the compiler messages from the NEC SX compiler at compilation of the Himeno benchmark [2]. From Lines 4 and 16 of the compiler messages, it can easily be identified that loops beginning from Lines 305 and 326 of *himeno.f90* are automatically parallelized. Furthermore, from Lines 13 and 14 of the compiler messages, it can be identified that the critical section for a summation calculation exists in Line 321 of the source code. As the above information can be obtained without reading any source codes, a programmer can focus on only inserting appropriate directives into the identified loops.

The proposed compiler-assisted OpenMP migration method uses compiler messages. The procedure consists of the following steps.

- Step 1. Compile a code using automatic optimizations in order to obtain compiler messages.
- Step 2. Identify parallelizable loop nests by checking the compiler messages.
- Step 3. Find variables that need to be listed in the reduction clauses of OpenMP directives. If a loop performs a reduction operation on a variable, the variable is a so-called reduction variable. For parallelizing such a loop with an OpenMP directive, all reduction variables must be listed in the reduction clause of the directive. A reduction variable can be found by looking for critical regions in the compiler messages.
- Step 4. Find variables that need to be listed in the private clauses of OpenMP directives. An array must be private if the value of its index depends on any loop index in the loop nest. In addition, a variable must be private if it is declared in the loop body.

Listing 1.1. A compiler messages from the NEC SX compiler

```

1:$ sxf90 -Pauto -Chopt -R2 -Wf,-pvctl fullmsg,-L objlist,summary himeno.f90
2:...
3:mul(10): himeno.f90, line 305: Parallel routine generated : jacobi$1
4:mul(1): himeno.f90, line 305: Parallelized by PARDO.
5:vec(1): himeno.f90, line 307: Vectorized loop.
6:vec(29): himeno.f90, line 307: ADB is used for array.: a
7:vec(29): himeno.f90, line 307: ADB is used for array.: bnd
8:vec(29): himeno.f90, line 307: ADB is used for array.: wrk1
9:vec(29): himeno.f90, line 307: ADB is used for array.: c
10:vec(29): himeno.f90, line 307: ADB is used for array.: b
11:vec(29): himeno.f90, line 307: ADB is used for array.: p
12:vec(29): himeno.f90, line 307: ADB is used for array.: wrk2
13:mul(4): himeno.f90, line 321: CRITICAL section.
14:vec(26): himeno.f90, line 321: Macro operation Sum/InnerProd.
15:mul(10): himeno.f90, line 326: Parallel routine generated : jacobi$2
16:mul(1): himeno.f90, line 326: Parallelized by PARDO.
17:opt(1057): himeno.f90, line 326: Complicated uses of variable inhibits
   loop optimization.
18:opt(1592): himeno.f90, line 326: Outer loop unrolled inside inner loop.
19:vec(4): himeno.f90, line 326: Vectorized array expression.
20:vec(3): himeno.f90, line 326: Unvectorized loop.
21:vec(13): himeno.f90, line 326: Overhead of loop division is too large.
22:vec(4): himeno.f90, line 326: Vectorized array expression.
23:vec(29): himeno.f90, line 326: ADB is used for array.: p
24:...

```

Step 5. Insert appropriate OpenMP directives by adding necessary clauses into the source code.

The detailed processes of the proposed OpenMP migration method using an example of the Himeno benchmark are explained. Listing 1.2 shows an OpenMP version of the Himeno benchmark, in which OpenMP directives are inserted according to the proposed method. In Step 1, the serial version of the Himeno benchmark is compiled to obtain its compiler messages shown in Listing 1.1. In Step 2, by using the compiler messages, parallelizable loops are identified. From messages “Parallelized by PARDO” in Lines 4 and 16 of Listing 1.1, the parallelizable loop nests can be easily identified. Then, in Step 3, a reduction operation at Line 321 can be detected from the compiler messages “CRITICAL section.” in Line 13 of Listing 1.1. Then, the variable, *GOSA*, is treated as a reduction variable to be listed in the reduction clause. Then, in Step 4, the local variables *S0* and *SS* in Lines 308 and 320 in the parallelized loop are decided as private variables because these variables are used only in the parallelized loop. Then, these variables are listed in an OpenMP private clause. Finally, OpenMP directives with the reduction and private clauses are inserted into the parallelizable loop nests based on the identifications of the parallelizable loop nests and their analyses. For the loop nest from Line 305 to Line 325, the OpenMP parallel directive with private and reduction clauses is inserted. For Line 326, the OpenMP workshare directive is inserted. By the procedure of the proposed method, the OpenMP version of the Himeno benchmark based on compiler messages can be generated.

The procedure of the proposed OpenMP migration method is useful to develop migration assistant tools and/or automatic migration tools. By developing such

Listing 1.2. Migrated OpenMP code by the compiler-assisted migration method

```

303:   do loop=1,nn
304:     gosa= 0.0
NEW: !$omp parallel do private(s0,ss), reduction(+:GOSA)
305:       do k=2,kmax-1
306:         do j=2,jmax-1
307:           do i=2,imax-1
308:             s0=a(I,J,K,1)*p(I+1,J,K) &
309:               +a(I,J,K,2)*p(I,J+1,K) &
310:               +a(I,J,K,3)*p(I,J,K+1) &
311:               +b(I,J,K,1)*(p(I+1,J+1,K)-p(I+1,J-1,K)) &
312:                 -p(I-1,J+1,K)+p(I-1,J-1,K)) &
313:               +b(I,J,K,2)*(p(I,J+1,K+1)-p(I,J-1,K+1)) &
314:                 -p(I,J+1,K-1)+p(I,J-1,K-1)) &
315:               +b(I,J,K,3)*(p(I+1,J,K+1)-p(I-1,J,K+1)) &
316:                 -p(I+1,J,K-1)+p(I-1,J,K-1)) &
317:               +c(I,J,K,1)*p(I-1,J,K) &
318:               +c(I,J,K,2)*p(I,J-1,K) &
319:               +c(I,J,K,3)*p(I,J,K-1)+wrk1(I,J,K)
320:     ss=(s0*a(I,J,K,4)-p(I,J,K))*bnd(I,J,K)
321:     GOSA=GOSA+SS*SS
322:     wrk2(I,J,K)=p(I,J,K)+OMEGA *SS
323:   enddo
324: enddo
NEW: !$omp end parallel
325: enddo
NEW: !$omp workshare
326:   p(2:imax-1,2:jmax-1,2:kmax-1)= &
327:     wrk2(2:imax-1,2:jmax-1,2:kmax-1)
NEW: !$omp end workshare
328: enddo

```

tools, higher productivity of the OpenMP programming is expected. As a result, a programmer focuses on writing contents of a code. Furthermore, as compiler messages completely depend on individual compilers, more useful information to migrate serial codes to the OpenMP model may be obtained by collecting compiler messages from multiple compilers. From these compiler messages, the characteristics of automatic parallelized codes by each compiler are analyzed, and more appropriate OpenMP parallelization can be employed for the target system.

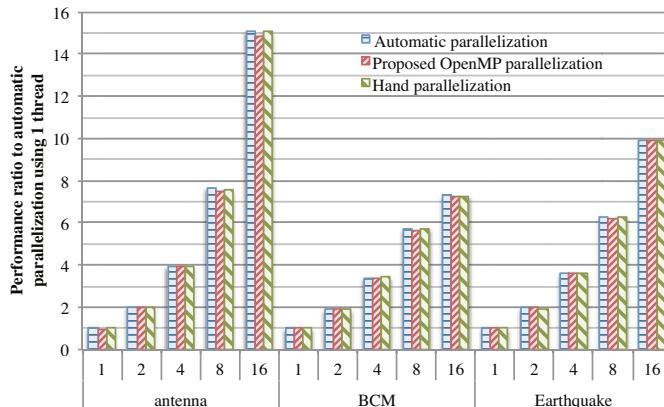
3 Performance Evaluation

3.1 Experimental Environments

In order to demonstrate the effectiveness of the proposed compiler-assisted method, three kernels of practical applications are migrated based on the proposed method. The Antenna kernel is for the FDTD simulation of a lens antenna [3] and its Bytes/Flop ratio is 1.73. The BCM kernel is the main calculation of CFD with equally-spaced Cartesian meshes [4] and its Bytes/Flop is 8.0. The Earthquake kernel is for simulation of the seismic slow slip model [3] and its Bytes/Flop ratio is 2.0. As these applications have been developed and optimized for NEC SX-9, the compiler messages obtained by the NEC SX compiler are utilized in the following evaluation.

Table 1. Node Specification of HPC systems used in the evaluations

Systems	Sockets	Cores	Compiler, options, and version
NEC SX-9	16	1	sxf90 -Popenmp/-Pauto -Chopt -R2 (Rev.460)
Intel Nehalem EX	4	8	ifort -openmp/-parallel -xHost -O3 (Version 12.1.0 20110811)
Fujitsu FX10	1	16	frtpx -Kopenmp,noparallel/-Kparallel,noopenmp -Kfast (Version 1.2.1)
Hitachi SR16000 M1	4	8	f90 -omp/-parallel -64 -model=M1 (Hitachi Optimizing FORTRAN90 Compiler)

**Fig. 2.** Performances of three kernels on NEC SX-9

In addition, the performances of the OpenMP codes developed with the proposed migration method are evaluated on multiple HPC systems to discuss their performance portabilities. The four HPC systems whose specifications are shown in Table 1 are used for the performance evaluations. The performances of a migrated OpenMP code by the proposed method, a hand-parallelized code, and its original serial code are compared using the HPC systems. The original serial code is migrated by using the compiler messages of the NEC SX compiler. The migrated OpenMP code is compiled by the compiler on each HPC system with OpenMP options as shown in Table 1. The hand-parallelized code is implemented manually by an expert programmer without any hints from compilers. Several optimizations such as maximizing parallel regions and selecting the best OpenMP scheduling method are applied to the hand-parallelized OpenMP codes. The original serial code is automatically parallelized by the compiler with enabling the highest-level optimizations except for the OpenMP support.

3.2 Verification of the Compiler-Assisted OpenMP Migration Method

Figure 2 compares the performances of SX-9 for three kernels, each of which has the original version, its proposed parallelization version, and its hand

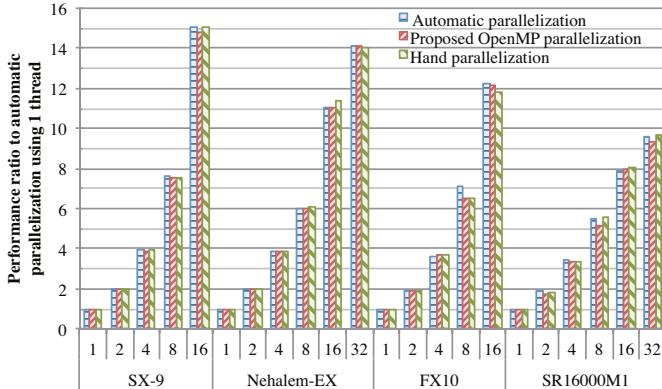


Fig. 3. Performance of the Antenna kernel among multiple HPC systems

parallelization version. The horizontal-axis of the figure indicates the kernels and the number of threads used in the evaluation. The vertical-axis indicates the sustained performances that are normalized by the performance of the original code executed on each HPC system with a single thread.

This figure shows that the performances of the original version, the proposed parallelization version, and the hand parallelization version of three kernels are comparable irrespective of any thread counts. Since the proposed OpenMP parallelization is performed based on the messages from the NEC SX compiler, almost the same optimizations are applied to the proposed OpenMP version.

In addition, the proposed OpenMP parallelization can achieve comparable performances to the hand parallelization. This is because the NEC SX compiler can successfully identify the parallelizable loop nests although an overhead due to creating and deleting a team of threads can slightly be observed. All loops that are able to be parallelized and/or vectorized in three kernels can be detected by the NEC SX compiler. Some of these loops are multithreaded, and the others are vectorized. Thus, the parallelized loop nests in the OpenMP version migrated by the proposed method are almost the same as the hand-parallelized ones. Therefore, these results demonstrate that the proposed method can successfully migrate the serial kernels to the OpenMP model.

3.3 Performance Portability Evaluation of Migrated OpenMP Codes

Figure 3 shows the evaluation results to compare the performances with the Antenna kernel code and its OpenMP versions on each HPC system. The horizontal-axis of the figure indicates the HPC systems and the number of threads used in evaluation. The vertical-axis indicates the sustained performances that are normalized by the performance of the original code executed on each HPC system with a single thread. This figure shows that the performances of the migrated OpenMP code are comparable to those of the original version on every HPC

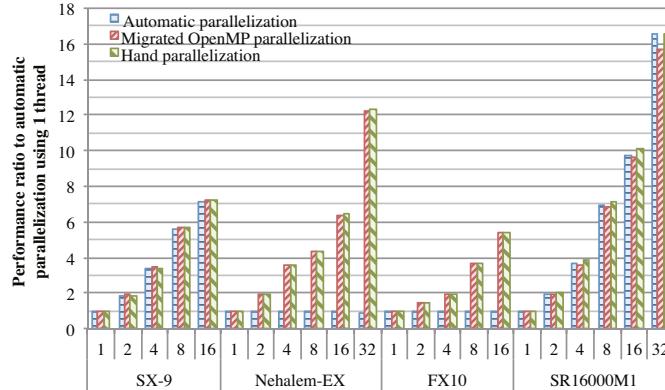


Fig. 4. Performance of the BCM kernel among multiple HPC systems

system. In addition, the performances of these codes are similarly improved as the number of threads increases. In the case of SX-9, the performances of these versions increase according to the number of threads. In the cases of the other HPC systems, the Antenna code is automatically parallelized by their own compilers. All loops that can be parallelized and/or vectorized are identified by the FX compiler. 83% of the loops are identified by the Intel and Hitachi compilers. Thus, the code is parallelized in the similar way as the inserted OpenMP directives do. Therefore, the migrated OpenMP code achieves a comparable performance to the automatically-parallelized code even on the other HPC systems. Furthermore, the performances of the migrated code by the proposed method are comparable to those of the hand parallelization. An overhead due to creating and deleting a team of threads can be observed in the performances of the migrated code by the proposed method because the parallel directive is simply inserted into every detected parallel loop nest. However, the overhead is sufficiently small.

Figure 4 shows the evaluation results to compare the performances with the BCM kernel code and its OpenMP versions on each HPC system. This figure shows that, in the cases of SX-9 and SR16000M1, the performance difference between the automatically-parallelized code and the migrated OpenMP code is always small. As with the case of the Antenna kernel, the compilers of SX-9 and SR16000M1 perform similar automatic parallelization. 84.6% of the parallelizable and vectorizable loops can be identified by the compiler of SR16000M1. In the cases of Nehalem-EX and FX10, the automatically-parallelized version cannot achieve a comparable performance to the OpenMP version. This is because their compilers are unable to identify most of parallelizable loop nests. Only 15.4% of the loops can be identified by their compilers since the dependencies among iterations cannot be analyzed. As the NEC SX compiler can find those loop nests, OpenMP parallelization based on the compiler message can parallelize the loop nests, and improve the performances of Nehalem-EX and FX10 as well as SX-9. These results clearly indicate that the message from the target

compiler is helpful for OpenMP parallelization. Since other compilers may not be able to find those parallelizable loop nests, OpenMP parallelization with the target compiler message is promising to improve the performance portability across different systems.

4 Related Work

Larsen et al. have proposed an interactive compilation feedback system by using compiler analyses to enhance the automatic parallelization capability of a compiler [5]. The system analyzes why a compiler does not parallelize a loop nest. Then, the system shows the reasons and the guidelines for modifying the code. A programmer modifies the code based on compiler analyses so that the modified code can further be optimized by the compiler. As the system lets a compiler optimize the code, more compiler messages of automatic optimizations can be obtained. These obtained compiler messages can provide even the proposed OpenMP migration method more clues for the OpenMP parallelization.

Many source translators that can automatically insert OpenMP directives have been developed such as Cetus[6], Rose[7], and Pluto[8]. These compiler frameworks are equipped with the parallelizing transformation functions, such as parallelizable region detection and private and reduction variable recognition functions, for automatic parallelization. The proposed OpenMP migration method basically needs programmer's helps to identify private and reduction variables since compiler messages do not always suggest private and reduction variables. Therefore, by combining private and reduction variable recognition functions of such compiler frameworks with the proposed OpenMP migrations, the burden of OpenMP migration can further be reduced. In addition, as the proposed OpenMP migration method can utilize compiler messages of any compiler, the compiler messages by commercial compilers and such compiler frameworks are also valuable for the proposed OpenMP migration.

5 Conclusions

Many HPC codes are manually modified so that the automatic optimizations of a particular target compiler can work efficiently. These manual optimizations help the target compiler generate more efficient codes. However, the performances of such HPC codes are not always portable to other systems with different compilers.

In order to enhance the performance portability, this paper proposes an OpenMP migration method that utilizes compiler messages of automatic optimizations from the target compiler. These messages including how the compiler performs automatic optimizations are helpful even for the optimizations on other systems. By using compiler messages, the proposed compiler-assisted OpenMP migration method realizes high performance portability with low programming efforts.

Experiments show that the proposed migration method can successfully migrate serial codes to OpenMP compilers with minimum efforts. In addition, it

is clarified that the messages from the target compiler is valuable for OpenMP parallelization. Since other compilers cannot always identify parallelizable loop nests, the OpenMP migration method with the target compiler message is effective to improve the performance portability across different systems.

Our future work includes the extensions of the proposed compiler-assisted migration method to other directives such as OpenACC and Intel MIC pragmas. The extensions can help migrate the original code to accelerators such as GPUs and Intel Xeon Phi. Moreover, an automatic OpenMP parallelization based on the proposed method can further reduce the efforts for the migration, which will also be addressed in our future work.

Acknowledgments. This research was partially supported by Grant-in-Aid for Scientific Research (S) #21226018 and Core Research of Evolutional Science and Technology of Japan Science and Technology Agency (JST CREST) “An Evolutionary Approach to Construction of a Software Development Environment for Massively-Parallel Heterogeneous Systems”.

References

1. Mustafa, D., Aurangzeb, Eigenmann, R.: Performance analysis and tuning of automatically parallelized openMP applications. In: Chapman, B.M., Gropp, W.D., Kumaran, K., Müller, M.S. (eds.) IWOMP 2011. LNCS, vol. 6665, pp. 151–164. Springer, Heidelberg (2011)
2. Himeno benchmark, <http://accr.riken.jp/2444.htm>
3. Soga, T., Musa, A., Shimomura, Y., Egawa, R., Itakura, K., Takizawa, H., Okabe, K., Kobayashi, H.: Performance evaluation of nec sx-9 using real science and engineering applications. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, pp. 1–12 (November 2009)
4. Nakahashi, K.: High-density mesh flow computations with pre-/post-data compressions. In: AIAA Paper, pp. 2005–4876 (2005)
5. Larsen, P., Ladelsky, R., Lidman, J., McKee, S.A., Karlsson, S., Zaks, A.: Parallelizing more loops with compiler guided refactoring. In: 2012 41st International Conference on Parallel Processing (ICPP), pp. 410–419 (September 2012)
6. Dave, C., Bae, H., Min, S.-J., Lee, S., Eigenmann, R., Midkiff, S.: Cetus: A source-to-source compiler infrastructure for multicores. IEEE Computer 42(12), 36–42 (2009)
7. Liao, C., Quinlan, D.J., Willcock, J.J., Panas, T.: Extending automatic parallelization to optimize high-level abstractions for multicore. In: Müller, M.S., de Supinski, B.R., Chapman, B.M. (eds.) IWOMP 2009. LNCS, vol. 5568, pp. 28–41. Springer, Heidelberg (2009)
8. Bondhugula, U., Baskaran, M., Krishnamoorthy, S., Ramanujam, J., Rountev, A., Sadayappan, P.: Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In: Hendren, L. (ed.) CC 2008. LNCS, vol. 4959, pp. 132–146. Springer, Heidelberg (2008)

A Type-Oriented Graph500 Benchmark

Nick Brown

EPCC, Edinburgh University, UK

nick.brown@ed.ac.uk

Abstract. Data intensive workloads have become a popular use of HPC in recent years and the question of how data scientists, who might not be HPC experts, can effectively program these machines is important to address. Whilst using models such as Partitioned Global Address Space (PGAS) is attractive from a simplicity point of view, the abstractions that these impose upon the programmer can impact performance. We propose an approach, type-oriented programming, where all aspects of parallelism are encoded via types and the type system which allows for the programmer to write simple PGAS data intensive HPC codes and then, if they so wish, tune the fundamental aspects by modifying type information. This paper considers the suitability of using type-oriented programming, with the PGAS memory model, in data intensive workloads. We compare a type-oriented implementation of the Graph500 benchmark against MPI reference implementations both in terms of programmability and performance, and evaluate how orienting their parallel codes around types can assist in the data intensive HPC field.

Keywords: Graph500, Mesham, type-oriented programming, data intensive workload, PGAS.

1 Introduction

The HPC community has traditionally concentrated on solving computation based problems but in recent years data intensive workloads have also become a popular use of these resources. Data intensive workloads often involve huge amounts of data, each requiring small numbers of calculations per element and because of this the communication aspects of a system is critically important. This is in contrast with the more traditional computation based workloads, where data sizes tend to be smaller but much more computation per element is required. One of the challenges associated with HPC is programming models and the data processing field is no exception. There is often a trade off between programmability and performance; those models which promote simplicity can impose choices and restrictions upon the programmer in the name of abstraction which can harm performance. Whilst efficient communication is important to both computation and data intensive workloads, the fact that data intensive workloads place so much emphasis on communication makes it critically important that the programming models used do not sacrifice communication efficiency.

Using the PGAS memory model for solving data intensive problems, where data is equally accessible whether it is held in local or remote memory and the programmer need not worry about the underlying implementation detail, is an attractive proposition from a programmability point of view. However, this higher level of abstraction can typically result in a performance cost and existing PGAS languages either disallow or limit the options that the programmer has to tune their code at the communication level. Type-oriented programming addresses the PGAS trade off issue by providing the options to the end programmer to choose between explicit and implicit parallelism by using types which can be combined to form the semantics of data, governing parallelism. A programmer may choose to use these types or may choose not to use them and in the absence of type information the compiler will use a well-documented set of default behaviours. Additional type information can be used by the programmer to tune or specialise many aspects of their code which guides the compiler to optimise and generate the required parallelism code. In short these types for parallelisation are issued by the programmer to instruct the compiler to perform the expected actions during compilation and in code generation.

The Graph500[1] benchmark is a popular, objective, way of determining hardware's suitability to data processing but can also be used to benchmark other aspects such as the languages, libraries and runtimes that are used in data intensive work. We compare an implementation of the Graph500 benchmark in Mesham, our research type-oriented programming language, compared to the MPI reference versions both in terms of programmability and performance.

2 Background

2.1 Type-Oriented Programming

Type-oriented programming[2] allows the programmer to encode all variable information via the type system by combining different types together to form the overall meaning of variables. This is contrasted against a more traditional approach where the programmer uses a type to govern the data that a variable will hold but additional information, such as whether a variable is read only or not, is applied via type qualifiers. Using the C programming language as an example, in order to declare a variable *m* to be a read only character where memory is allocated externally, the programmer writes *extern const char m*. Where *char* is the type and both *extern* and *const* are inbuilt language type qualifiers. Whilst this approach works well for sequential languages, in the parallel programming domain there are potentially many more attributes which might need to be associated; such as where the data is located, how it is communicated and any restrictions placed upon this. Representing such a rich amount of information via multiple qualifiers would not only bloat the language, it might also introduce inconsistencies when qualifiers were used together with potentially conflicting behaviours.

Instead our approach is to allow for the programmer to combine different types together to form the overall meaning. For instance, *extern const char m* becomes

var m:Char::const::extern, where *var m* declares the variable, the operator *:* specifies the type and the operator *::* combines two types together. In this case, a **type chain** is formed by combining the types *Char*, *const* and *extern*. Precedence is from right to left where, for example, the read only properties of the *const* type override the default read & write properties of *Char*. It should be noted that some type coercions, such as *Int::Char* are meaningless and so rules exist within each type to govern which combinations are allowed.

Within type-oriented programming the majority of the language complexity is removed from the core language and instead resides within the type system. The types themselves contain specific behaviour for different usages and situations. The programmer, by using and combining types, has a high degree of control which is relatively simple to express and modify. Additionally, by writing code in this high level way means that there is a rich amount of information upon which the compiler can use to optimise the code. In the absence of detailed type information the compiler can apply sensible, well documented, default behaviour and the programmer can further specialise this using additional types if required at a later date. The result is that programmers can get their code running and then further tune if needed by using additional types.

Mesham[3] is a programming language that we have developed to research and evaluate the type-oriented paradigm. It follows a simple imperative language with extensions to support this type-oriented paradigm and a Partitioned Global Address Space memory model. Using the PGAS memory model, the entire global memory, which is accessible from every process, is partitioned and each block has an affinity with a distinct process. Reading from and writing to memory (either local or another processes' chunk) is achieved via normal variable access and assignment. The benefit of PGAS is its conceptual simplicity, where the programmer need not worry about the lower level and often tricky details of communication. This makes it an ideal memory model for data scientists, who are experts in using their own data but not necessarily HPC.

Type-oriented programming provides the best of both worlds. In Mesham by default all communication is one sided however this can be overridden using additional type information which further tunes and specialises the communication behaviour of specific variables. The programmer can get their codes working and then tune, by using types, fundamental aspects such as communication to improve performance and/or scalability. This is contrasted against traditional HPC programming models, where fundamental aspects are often integral to the code and changing them can require widespread changes or even a code rewrite.

2.2 Graph 500 Benchmark

As data intensive workloads are fundamentally limited by communication, existing computation based benchmarks such as LINPACK are of limited use. Instead, the Graph500 benchmark[1] has been developed to stress the communication aspects of a system. It consists of three phases; graph construction which constructs graph in Compressed Sparse Row (CSR) format, a Breadth First Search (BFS) kernel and lastly the a validation of the BFS traversal.

The Graph500 problem size is represented using scale and edge factors. The scale is the logarithm base two of the number of vertices and edge-factor is the ratio of the graphs edge count to its vertex count. Therefore a graph has 2^{scale} vertices and $2^{\text{scale}} \times \text{edge-factor}$ edges. Performance of the BFS is measured in Traversed Edges Per Second (TEPS.) A number of reference implementations are provided and of these there are four MPI based codes, a one-sided implementation where vertex communications are done using MPI-2 one-sided communications, an MPI simple version which uses asynchronous point to point communications, an MPI replicated compressed sparse row implementation and a MPI replicated compressed sparse column code. Whilst all of these implementations use the level synchronized BFS traversal algorithm [4] their implementations all require separate codes and to go from one to another has required substantial code rewriting of the BFS kernels. In this paper we concentrate on the one-sided and the simple (asynchronous point to point) versions which, at an algorithm level, only differ in terms of their form of communication but require entirely separate kernel implementations. In reality data scientists do not want to be working at this lower level; they want to be able to write a simple code and then easily modify fundamental aspects, such as the form of communication, to experiment with parallelism.

2.3 Related Work

High Performance Fortran (HPF)[5] is a parallel extension to Fortran90. The programmer specifies just the data partitioning and allocation, with the compiler responsible for the placement of computation and communication. The type-oriented approach differs because programmer can, via types, control far more aspects of parallelism. Alternatively, if not provided, the type system allows for a number of defaults to be used instead. Co-array Fortran (CAF)[6] provides the programmer with a greater degree of control than in HPF, but still the method of communication is implicit and determined by the compiler whilst synchronisations are explicit. CAF uses syntactic shorthand communication commands and synchronisation statements hard wiring these into a language is less flexible than our use of types. Chapel[8] is another PGAS language and supports the programmer controlling aspects of parallelism by providing higher and lower levels of abstractions. Many of these higher level constructs in Chapel, such as reduction are implemented via inbuilt operators and keywords, contrasted to Mesham where they would be types in an independent library.

Co-array C++ [7] integrates co-arrays into C++ using template libraries. The C++ programmer adds additional information to their source code through these template libraries which determine parallelism. Whilst the type library of Mesham has a far wider scope than the current co-array template library it would be possible to encode our types as a C++ template library. This illustrates how the core language itself is actually irrelevant and our approach could be applied to existing languages, such as C++. The benefit of this is that programmers could orient their parallelism around types within a familiar language. The downside of this approach is that the actual C++ language is fixed and whilst template

libraries are well integrated, the flexibility of our use of types would be limited to the current C++ approach and compile time optimisation might be limited.

Parallelizing ARRAYS (PARRAY)[9] extends the C and C++ languages with new typed arrays that contain additional information such as the memory type, layout of data and the distribution over multiple memory devices. The central idea is that a programmer need only learn one unified style of programming and this applies equally to all major parallel architectures. The compiler will generate code according to the typing information contained in the source. Whilst this approach is similar to the types used in Mesham, there are some important differences. As a bolt on to existing languages, PARRAY uses its own syntax, similar to pre-processor directives, to declare arrays with types; for instance *pinned*, *paged* or *dmem* are used denote which device holds the array. In dealing with the arrays PARRAY still requires a number of inbuilt commands to handle its data structures. Mesham takes this a stage further and types are fully integrated into the language which means that, instead of requiring commands to copy or transpose data, language operators such as assignment will automatically handle the operation according to the type information. This integration is central to the data intensive example considered here where parallelism is entirely integrated in the language; for instance references to data may be local or global but in the simplest case the programmer need not worry about distinguishing these aspects to get their code working. In our approach there many types which the programmer can use to tune their data structures but equally omitting these is fine which will result in some safe default behaviour being applied.

The approach that PARRAY follows, bolting on parallelism using some syntax to differentiate it from the existing language is familiar. Solutions such as OpenMP[10] allow for the programmer to direct parallelism through pre-processor directives which guide the compiler how to handle parallelism. Importantly in our approach, types are first class citizens in the language so integrate fully with the existing language semantics which means that the programmer has the flexibility to support aspects such as creating new types in their code and reasoning about type information using existing language constructs. Through constructing type chains we provide a mechanism for building up complex type information in a structured manner and it is this type chain that provides the semantics of operations performed on the variable.

3 Implementation

Listing 1.1 illustrates the Mesham source code for the BFS kernel and associated variable declarations. For clarity we concentrate on the core BFS kernel, hence supporting functions such as building the Kronecker graph of vertices, constructing the edge list, finding search keys and validating the resulting search tree have been omitted from this paper. The *typevar* keyword at line 1 creates a new type which is called a *GraphVertex* and is basically an alias for the *referencerecord* type which is a record and similar to a struct in C. This record is used to represent a graph vertex and the *referencerecord* type supports members of a record

referencing other records. An example of this is the array at line 1, *children*, which contains references to other vertex records which are the children of this vertex. The references themselves can either be to local data or point to global data which is held on another process. In terms of correctness, the programmer need not distinguish between local and global data references as the type library takes care of the underlying communications required; although they might want to differentiate for performance reasons.

```

1 typevar GraphVertex ::= referencerecord ["children", array [
2   GraphVertex], "numChildren", Long, "id", Long];
3 var vertices:array[GraphVertex, nvtx_scale] :: allocated [
4   partitioned [numProcs] :: single [evendist]]
5
6 var searchTree:array[Long, nvtx_scale] :: allocated [partitioned [
7   numProcs] :: single [evendist]];
8 var childrenParents:array[Long, nvtx_scale] :: allocated [
9   partitioned [numProcs] :: single [evendist]];
10 var globalNextVerticies:=1;
11 var vertexQueue:queue[GraphVertex] :: allocated [multiple];
12 var vertexQueueNext:queue[GraphVertex] :: allocated [multiple];
13
14 if (vertices [rootVertexIndex].on == myPid) { vertexQueue:=root;
15   childrenParents [root.id]:=root.id; };
16 while (globalNextVerticies > 0) {
17   while (!vertexQueue.empty) {
18     var singleVertex:GraphVertex;
19     singleVertex:=vertexQueue;
20     if (searchTree [singleVertex.id] == -1) {
21       searchTree [singleVertex.id]:=childrenParents [singleVertex.
22         id];
23       for i from 0 to singleVertex.numChildren - 1 {
24         var childVertex:=singleVertex.children [i];
25         childrenParents [childVertex.id]:=singleVertex.id;
26         vertexQueueNext.on [childVertex.on]:=childVertex;
27       }; };
28     vertexQueue:=vertexQueueNext;
29     vertexQueueNext.clear;
30     globalNextVerticies :: allreduce ["sum"]:=vertexQueue . size; };

```

Listing 1.1. Default one sided communication

Lines 2, 4 and 5 set up arrays *vertices*, the actual graph vertices which have already been built up, *searchTree*, the search tree to return from this kernel holding the resulting vertex parent ids and *childrenParents* which holds the parent ids of children vertices for the next level. Using the *allocated* type the programmer has provided some additional information to guide the compiler in how to allocate this data. Combined with the *single* type it means that a single copy of the array exists globally, which is split up into *numProcs* partitions which are then evenly distributed amongst the distinct process memories via the

evendist type. Because these three arrays are the same size they are distributed in the same manner with the same indexes (and hence vertices) on each process.

Lines 7 and 8 create two queues, one for the current BFS level and one for the next BFS level with each queue holding data of type *GraphVertex*. The use of the *multiple* type as an argument to the *allocated* type informs the compiler that each process will hold a distinct version of these variables in their own memory. Each type determines the behaviour whenever a variable is used, an example of this is at line 10 where the *referencerecord* type implements the *on* operator which returns the process which holds the actual data which the reference is pointing to. In this case the effect is to add the root search vertex to the current level queue on that specific owning process. The effect of the assignment at line 14 will be to pop the top most vertex from the local queue and place it into the *singleVertex* variable, the fact that this is a pop is because of the appropriate types of the variables involved in the assignment and for the same reason a queue addition is done at lines 20 and 10 but line 22 copies the entire *vertexQueueNext* into *vertexQueue*. For each level, a process will iterate through their vertex queue. For each vertex if it has not already been processed (line 15) then the process will iterate through each child. Each child is added to the next level vertex queue on the appropriate holding process as well as that child's parent id to the *childrenParents* array. In the absence of further type information, how these communications occur is entirely abstracted away and if the child is on a different process to the parent then the default behaviour will be one sided communication. Part of a global reference is which process's memory actually holds the data. This means that a *referencerecord*'s *.on* operation is a local operation on the reference itself and the vertex communication required in this BFS implementation is to place remote vertices on their own processes' next level queue; much of this remote placing can be implemented with minimal communication. At line 24 there is a global all-reduce to determine the number of vertices to be processed at the next level and the algorithm will terminate if, at the global level, there are none. This *globalNextVerticies::allreduce["sum"]* illustrates an additional aspect of type-oriented programming where the type behaviour of a variable can be overridden by the addition of extra types for a specific expression; for this assignment only a blocking all-reduce is issued.

It can be seen that this is a simple, high level algorithm with the underlying types taking care of much of the lower level and tricky implementation details. Compared to the existing one sided MPI reference code this implementation is far shorter, 28 lines of code compared to 205 in the reference implementation and allows the programmer to concentrate on the algorithmic and data structures of their code. It is true that there are underlying Mesham types and runtime libraries to support the code which amount to 500 lines of code, however, these are very general and can be reused in multiple codes; the partition and distribution types we used for this benchmark were originally written for a Mesham asynchronous Jacobi implementation [11]. It illustrates how an HPC expert can construct these types once and these then can be used time and time again in different contexts. By simplifying the code, a data scientist, who might not

be an HPC expert, will be able to get their code working and to a reasonable performance level. Whilst the default one sided communication is a simple and safe behaviour it is often not particularly efficient. As already mentioned, one of the MPI reference implementations is a point to point code, which replaces the one sided communication with asynchronous point to point and greatly improves the efficiency. However, to achieve this the core BFS code has had to be rewritten and additional low level issues such as matching asynchronous communications and buffer sizes which are complex and error prone has had to be considered. By orienting all aspects of parallelism around types the programmer can get their code working and then further tune for performance and listing 1.2 sketches the modified Mesham code of listing 1.1 to use asynchronous point to point communication rather than the default one-sided.

```

var childrenParents : array[Long, nvtx_scale] :: allocated [
    partitioned [numProcs] :: single [eventdist]] :: async;
var vertexQueue : queue[GraphVertex] :: allocated [multiple] :: 
    async;
var vertexQueueNext : queue[GraphVertex] :: allocated [multiple] :: 
    async;
...
while (globalNextVertices > 0) {
    while (!vertexQueue.empty) {
        ...};
    sync;
    vertexQueue := vertexQueueNext;
    ... };

```

Listing 1.2. Asynchronous p2p communications

The code structure has remained the same and minimal changes, mainly oriented around the types, have been made to modify the underlying communication method. The addition of the *async* type to the *childrenParents* variable and queues guide the compiler to use asynchronous point to point communication for all communications involving these variables. The assignment at line 19 (listing 1.1) now issues an asynchronous send of the parent vertex id and at line 20 an asynchronous send as part of the remote queue addition. The *sync* keyword has been added at the end of processing the current level in listing 1.2 and waits for all outstanding asynchronous communications to complete, ready to proceed with the next search level. Effectively the addition of these *async* types set up the same asynchronous message listening and coalescing buffers that are present in the reference MPI implementation but these low level details are abstracted away from the programmer. In the absence of further type information the size of the coalescing buffer is set to be 256 *GraphVertex* elements, although this can be further tuned by providing an argument to the *async* type such as *async[128]*. This is an example of where the meaning of the argument provided to types depends entirely on the type chain and it is within the context of, in this case, the *queue* and *async* types to interpret the arguments accordingly.

4 Results

Figure 1 illustrates the strong scaling characteristics of our Mesham and MPI BFS implementations on a Cray XE6 using a vertex scale of 29. The upper plots, labelled *p2p*, are the asynchronous Mesham point to point BFS implementation and the reference MPI simple implementation. It can be seen that the performance of the Mesham and MPI versions are comparable, with the Mesham version slightly under performing the MPI implementation but the difference is small. Initially in both versions the TEPS increases as the number of cores is increased but, commonly with strong scaling experiments, a point is reached where the cost of communication outweighs the benefits gained from additional parallelism and performance starts to degrade. In the results we can see that the MPI implementation actually performs worse over 8192 cores than on 4096 cores, and the Mesham version's TEPS at 8192 cores is only a slight improvement over the 4096 core run.

Figure 1 also depicts the strong scaling performance of the default, one sided, Mesham implementation and the one sided MPI benchmark which are the lower two plots and labeled *onesided*. This illustrates that the safe and simple behaviour incurs a performance hit which can then be tuned using additional types to the performance in the p2p case. The Mesham one sided implementation out performs the MPI one-sided implementation due to the compiler optimising communications and one sided epoch windows which is possible because of the rich amount of type information available. This illustrates, in itself, a performance benefit of writing high level data intensive codes using Mesham default communications compared to a lower level implementation.

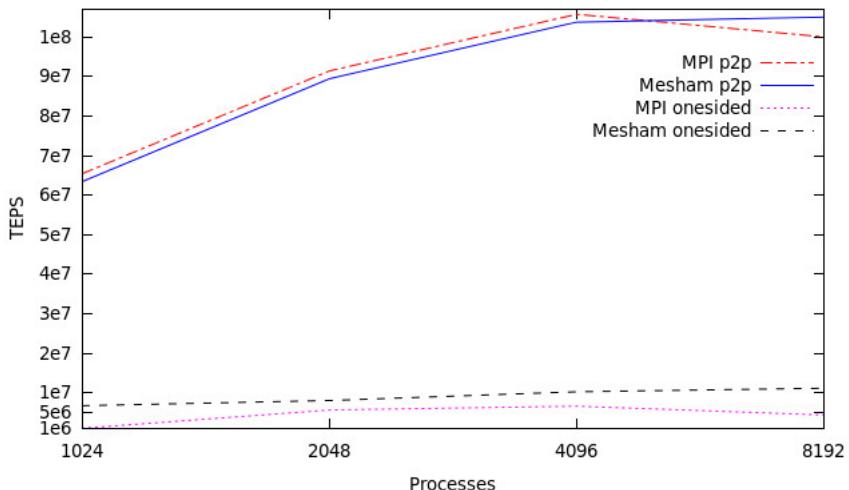


Fig. 1. Performance of Mesham vs MPI benchmarks

5 Conclusions and Further Work

In this paper we have considered how type-oriented programming may be applied to the data intensive HPC field. Aspects of this paradigm could, in the future, be used as part of existing languages to achieve the best of both worlds; the advantages discussed in this paper along with the familiarity of existing models and languages. We have shown that, by using types, the programmer can write conceptually simple PGAS style data processing codes at no significant hit in performance compared to traditional implementations. Types provide the additional benefit that the programmer can initially concentrate on the correctness of their codes and then, once a simple working version exists, they can use types to tune for performance as illustrated in the the Mesham one sided and point to point BFS implementations. There is further work to be done understanding the reasons behind the slight performance gap of the Mesham and MPI implementations and based upon this work we are now looking to examples of data intensive problems, rather than benchmarks, to understand how this paradigm can help in solving real world data intensive workloads.

References

1. Murphy, B., Wheeler, K., Barrett, B., Ang, J.: Introducing the Graph 500. In: Cray User's Group (CUG) (2010)
2. Brown, N.: Applying Type Oriented Programming to the PGAS Memory Model. In: 7th International Conference on PGAS Programming Models (PGAS) (2013)
3. Brown, N.: The Mesham language specification (2013), <http://www.mesham.com>
4. Suzumura, T., Ueno, K., Sato, H., Fujisawa, K., Matsuoka, S.: Performance Characteristics of Graph500 on Large-scale Distributed Environment. In: 2011 IEEE International Symposium on Workload Characterization, IISWC 2011 (2011)
5. Luecke, G., Coyle, J.: High Performance Fortran Versus Explicit Message Passing On The ISB SP-2. Technical Report Iowa State University (1997)
6. Numrich, R., Reid, J.: Co-array Fortran for parallel programming. ACM SIGPLAN Fortran Forum (1998)
7. Johnson, T.: Coarray C++. In: 7th International Conference on PGAS Programming Models (PGAS) (2013)
8. Cray Inc. Seattle: Chapel language specification (version 0.82) (2011), <http://chapel.cray.com/>
9. Chen, Y., Xiang, C., Hong, M.: PARRAY: A Unifying Array Representation for Heterogeneous Parallelism. In: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (2012)
10. OpenMP Architecture Review Board: OpenMP Application Program Interface Version 3.0 (2008), <http://www.openmp.org/mp-documents/spec30.pdf>
11. Brown, N.: Type oriented programming for exascale. In: Exascale Applications and Software Conference (EASC) (2013)

A Dynamic Execution Model Applied to Distributed Collision Detection

Matthew Anderson, Maciej Brodowicz, Luke Dalessandro,
Jackson DeBuhr, and Thomas Sterling

Center for Research in Extreme Scale Technologies, School of Informatics and
Computing, Indiana University, Bloomington, IN 47408, USA

Abstract. The end of Dennard scaling and the looming Exascale challenges of efficiency, reliability, and scalability are driving a shift in programming methodologies away from conventional practices towards dynamic runtimes and asynchronous, data driven execution. Since Exascale machines are not yet available, however, experimental runtime systems and application co-design can expose application-specific overhead and scalability concerns at extreme scale, while also investigating the execution model defined by the runtime system itself. Such results may also contribute to the development of effective Exascale hardware.

This work presents a case study evaluating a dynamic, Exascale-inspired execution model and its associated experimental runtime system consisting of lightweight concurrent threads with dynamic management in the context of a global address space examining the problem of mesh collision detection. This type of problem constitutes an essential component of many CAD systems and industrial crash applications. The core of the algorithm depends upon determining if two triangles intersect in three dimensions for large meshes. The resulting collision detection algorithm exploits distributed memory to enable extremely large mesh simulation and is shown to be scalable thereby lending support to the execution strategy.

1 Introduction

As the class of applications anticipated to require Exascale computing continues to grow, the need to address the key Exascale software challenges of efficiency and scalability continues to motivate changes in both conventional and emerging programming models. While Dennard scaling is coming to an end, hardware parallelism continues to grow exponentially thereby emphasizing medium and fine grain thread parallelism rather than the coarse-grained process parallelism favored by conventional practices. Emerging programming models can go beyond conventional practices by eliminating global barriers and exposing more parallelism moving from the coarse grained computation regime to medium and fine grained computation. A cross cutting dynamic execution strategy guides the overheads and contributes to network latency hiding through resource oversubscription. The dynamic execution strategy also enables a runtime system, via

event driven mechanisms, to exploit information not available at compile or load time to manage the unpredictable variability in resource access time. This work explores and tests the hypotheses of such a class of execution strategies using the problem of collision detection as the proxy application.

Collision detection is an important application in many industrial applications and in simulation software, including physics engines and CAD systems. It also comprises a scientific computing kernel for some applications requiring Exascale, including impulsive loading simulations for high velocity impact. Significant speedups in performance of parallel collision detection by using lightweight concurrent threads in a shared memory setting have been reported before. However, the relative value of a global address space model in conjunction with a dynamic execution strategy has never been investigated. Global address space models enable parallel collision detection on problem sizes much larger than the memory capacity of a single compute node (hereafter referred to as *locality*) while at the same time providing a scalable solution to an algorithm that is central to a wide variety of industrial applications. Using an open source runtime system developed to support this research, collision detection is explored using a Bounding Volume Hierarchy (BVH) with tree construction performed lazily in the context of a global address space and using a dynamic execution strategy.

Overall, this work provides the following new contributions:

- It provides a distributed collision detection algorithm based on a dynamic execution strategy.
- It provides distributed collision detection scaling results using a global address space model.
- It introduces an open source, C-language based, dynamic runtime system providing lightweight concurrent threads, local control objects for event driven synchronization, message driven style communication, and a partitioned global address space.

This work is structured as follows. Related work is provided in Section 2 while the implementation and hypotheses of the dynamic execution strategy are given in Section 3. The distributed collision detection algorithm implementation is given in Section 4 and the scaling and performance results are found in Section 5. Conclusions and future work are provided in Section 6.

2 Related Work

There are several efforts currently underway to expose more parallelism through dynamic task management in a wide variety of runtime systems and programming models. Some of the more prominent examples are the Intel Threading Building Blocks library (TBB), Cilk++, Chapel, Charm++, Unified Parallel C (UPC), HPX-3, XKaapi, and Coarray Fortran 2.0. Among these, Charm++, HPX-3, and UPC notably feature a global address space model like what is explored in this work. Starting several decades ago, many programming models began implementing some of the features of many-tasking explored in this case

study, including the Actors model [4], and Multilisp [6]. Cilk++ has been successfully used for multi-core collision detection in mechanical assemblies [3] using lightweight concurrent threads. Parallel approaches to collision detection which can construct and traverse the BVH include both GPU-based and CPU-based multi-core implementations [8,1]. However, these approaches are limited to the memory of a single locality and are not scalable to distributed systems. More recent efforts have utilized a global address space [5] for the computational science kernel but still use a collision detection algorithm limited to the memory of a single locality. The dynamic execution strategy presented here presents a distributed alternative to collision detection using event driven semantics and fine grain computation to hide network latency in the context of a global address space.

3 Dynamic Execution Strategy

The experimental ParalleX execution model [2] forms the core of the dynamic execution strategy for the collision detection approach explored here. It is designed to mitigate the key sources of performance decay identified by the SLOWER performance model [7], summarized as **Starvation**, or inadequate amount of work necessary to utilize available computation resources, **Latency**, or the time delay required to access remote resources, **Overhead**, or the additional work needed

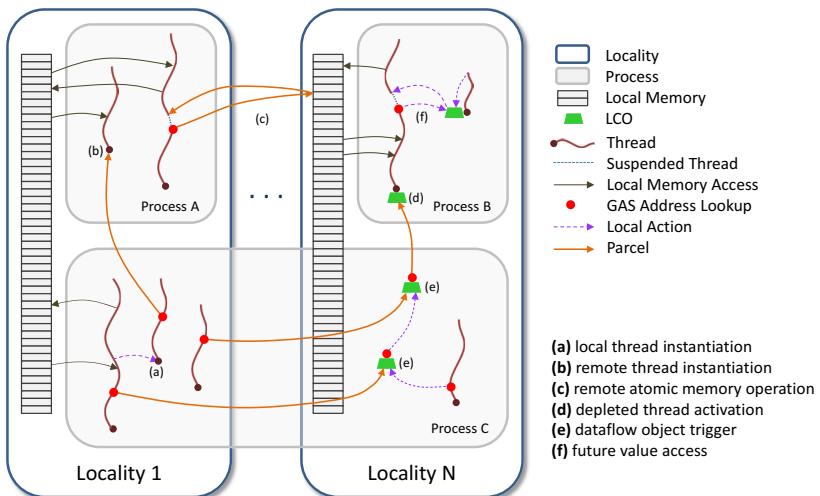


Fig. 1. Interaction of primary semantic components of the dynamic execution model. Local Control Objects (LCO) provide event coordination and management for the lightweight threads in the context of the Global Address Space (GAS). Six types of events are illustrated, (a)–(f), including examples of two specific types of LCOs, futures and dataflow. The execution strategy illustrates a work queue many-tasking execution model.

for resource allocation and management which is absent from sequential operation, **Waiting** for contention, or the time needed to resolve the contention on simultaneously accessed resources, **Energy efficiency**, and **Resilience**, or achieving reliable computation in the presence of faults.

The relationships between the primary semantic components of the execution model are illustrated in Figure 1. The fine- to medium-grain parallelism available in the application (tasks) is mapped onto a number of ephemeral user-level threads. These threads may access local memories in their localities of execution without any performance penalty; however, they are also capable of interacting with remote computational objects distributed throughout the system. The latter requires translation of the global address of the accessed object, shown as red dots in the diagram. If the object occupies the same locality as that of the requestor, the access is performed directly, thus incurring only a minor overhead. Otherwise, an active message (*parcel*) contains the operands and description of the function to be invoked is emitted to the destination locality, where the desired action is carried out using remote thread instantiation. The parcel targets may include explicit memory locations as well as synchronization objects (LCOs) that help manage the distributed control flow in the program. Besides the traditional locks or mutexes, they include higher level constructs such as *futures* [6] and *dataflow* blocks. Since the thread scheduler is aware of the potential for blocking when accessing synchronization objects or waiting for the parcel results, a high level of execution resource utilization and latency masking may be achieved as long as there is sufficient amount of work to be performed.

An open source runtime system has been developed to support research into this execution strategy. The HPX-5 runtime system is a user-level, multithreaded C language based implementation of ParalleX with local control objects such as futures for synchronization all within the context of a global address space.

4 Collision Detection

Mesh-based collision detection algorithms must simply decide if there exists a facet (i.e. triangle) a in mesh A and a facet b in mesh B such that A intersects with B , a test which could require a maximum of $O(n^2)$ facet collision tests. In practice, the number of tests that must be performed can be dramatically reduced by using a hierarchical partitioning of the bounding volume of the mesh, the aforementioned Bounding Volume Hierarchy (BVH), allowing a large number of facet collisions to be handled by a single bounding-box collision. BVH hierarchies may be reused over many related queries and thus parallelizing the BVH-based query algorithm is an ideal candidate for multi-threaded computation. This section describes the multi-threaded parallelization strategy used in this work, and the role of the global address space in that strategy.

The BVH in this work consists of a tree of progressively smaller axis-aligned bounding boxes. The root of the tree is the smallest bounding box—represented as a pair of points in three dimensions—that contains the entire mesh. Its children are bounding boxes that represent a complete partitioning of its space. This subspace partitioning continues until a threshold of constituent facets is reached.

The original, shared memory algorithm uses a traditional linked tree structure, with each node containing its points as well as pointers to its children. In this distributed implementation, the BVH is stored in the global address space in a flattened array containing indices that point to the children of a given node. In addition to the tree, the facet and point data from the meshes are stored in the global address space, along with the point data giving opposite corners of the bounding boxes. Again, these data are stored in an array where the index to the location in the array is sufficient to reconstruct the entire object.

The work of traversing the tree occurs distributed throughout the computational resources, so to simplify this process and reuse as much of the shared-memory implementation as possible, the tree node and facet objects are designed to be initialized with the index to their data in the global address space. In this way, only the portion of the tree relevant to an active computation may be reconstructed locally in a lazy, on-demand fashion. As an example, consider creating a local copy of a node from the global address space. Given only the index, the local node is constructed in a hollow form containing only that index. As soon as a method is invoked on that node that requires the full set of data, the local object pulls the information from the global address space. In particular, the node will create its children in the same hollow form.

The BVH data is saved in the global address space and the traversal of the tree is distributed across the available processing elements in the following way, as illustrated in Figure 2. Consider a comparison of two nodes, one from each BVH. The bounding boxes of the two nodes in consideration are tested, and if they overlap, the traversal proceeds to the children of the node with the largest bounding box. New lightweight user-level threads are spawned for each child of the opened node and the comparison of those children to the unopened node are performed by those threads (see label 1 in Figure 2). The lightweight threads spawned in this process, and the others described below, are distributed randomly among the available localities. The return values of these threads then are reduced via a logical or operation and returned (label 4 in Figure 2). This process is then repeated in the new threads of execution leading to more threads. Synchronization of these threads is achieved using futures, so a parent thread will not return until all threads it has spawned return.

Eventually leaves of the BVH are reached (label 2 in Figure 2) and the collision detection proceeds with the facets represented by the BVH leaves in question. Each facet in one leaf must be tested against each facet from the other. This may be parallelized by launching a number of lightweight threads, each responsible for performing the collision test between two particular facets (label 3 in Figure 2). Such fine-grained parallelism is expected to be important for managing latency and work imbalance in Exascale systems, but given the size of the test system and the choice to randomly distribute lightweight threads across localities, latency and work imbalance were not found to be significant for this work, so leaf collisions are executed with course-grained—but still lightweight—threads. This decision will be revisited for tests on larger systems.

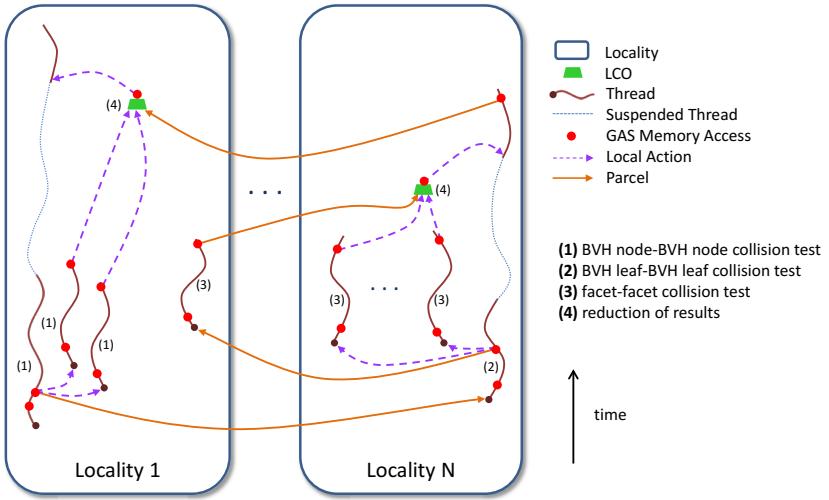


Fig. 2. The distributed implementation of the collision detection algorithm using a dynamic execution model. Four separate aspects of the algorithm are illustrated, including node-node collision detection, leaf-leaf collision detection, facet-facet collision detection, and reduction of results. GAS memory access, indicated by a red dot, indicates where the global address space containing mesh and BVH information is queried. Currently, GAS memory accesses are satisfied with synchronous RDMA get/put operations.

The full collision detection then begins with a single lightweight thread starting the detection by comparing the two root nodes of the two BVH. As the traversal proceeds, a large number of threads are spawned until eventually some threads are comparing the lists of facets in two leaf bounding boxes.

5 Results

The scaling results for the distributed collision detection strategy introduced in Section 4 are presented here for two CAD-generated, non-deformable, intersecting meshes containing over 130,000 vertices and 260,000 faces each with intersections in over 17,000 faces. Simulations were conducted on an Infiniband enabled cluster consisting of 16 compute nodes containing dual 8-core E5-2670 Intel processors running at 2.6 GHz with 32 GB of memory each. Mesh collision results were validated against an independent, serial collision detection code.

Figure 3 shows the collision detection performance on a single locality using up to 16 cores and showing a speed-up of 8.8. Figure 4 shows the performance on several localities where the number of operating system threads per locality was varied. The network latency in memory access causes a significant slow down compared to the single locality cases. Access to the global address space was synchronous making the overlap of communication with computation to hide

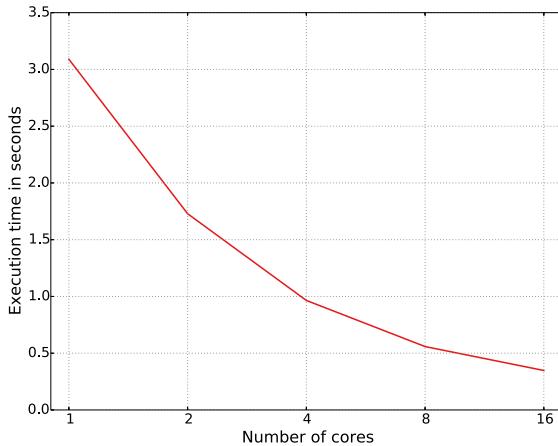


Fig. 3. Single locality scaling results for the collision detection code. The benchmark was run on a 16-core SMP machine, achieving a speed-up of 8.8.

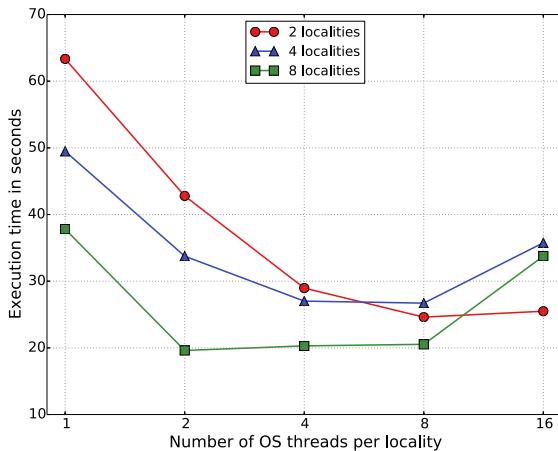


Fig. 4. Distributed scaling results of the collision detection code. The results were obtained on a Linux cluster equipped with 16-core localities. The code demonstrates good initial scaling behavior when increasing the number of OS threads for any number of localities tested. The best performing case on 8 localities performed 3.2× better than the slowest case on 2 localities.

network latency difficult. The best performing case on 8 localities performed 3.2 times better than the slowest case on 2 localities. The exact same code is used on multiple localities as on a single locality; no coding alterations are needed for distributed execution. For distributed cases, the maximum size limit of the meshes is no longer bounded not by the memory of a single locality and is scalable by the number of localities available in the system.

6 Conclusions

This case study has examined the problem of distributed collision detection using a dynamic execution model with a global address space which integrates the entire system stack. The approach represents a shift from conventional practice towards a more dynamic, data-driven approach using a runtime system

which exploits information not otherwise available at compile or load time. A distributed collision detection strategy was also introduced and implemented based on lightweight concurrent threads and event driven semantics. This strategy showed scalability on both a single locality and multiple localities. However, on multiple localities there was a significant performance decay due to network latency.

Future Work. While access to the global address space was synchronous in this case study, asynchronous global address space would better enable the overlap of communication with lightweight-thread-based computation. Because ParalleX also incorporates the ability to use GPUs within the global address space via percolation, thereby enabling the offloading of key collision detection computations to the GPU, performance improvements may result from using distributed hybrid CPU/GPU collision detection scheme. Other future improvements include reducing the number of LCOs used in the algorithm by eliminating the branch-and-reduce tree traversal approach.

References

1. Damkjaer, J., Erleben, K.: GPU accelerated tandem traversal of blocked bounding volume hierarchy collision detection for multibody dynamics. In: Prautzsch, H., Schmitt, A., Bender, J., Teschner, M. (eds.) VRIPHYS, pp. 115–124. Eurographics Association (2009)
2. Dekate, C., Anderson, M., Brodowicz, M., Kaiser, H., Adelstein-Lelbach, B., Sterling, T.: Improving the scalability of parallel N-body applications with an event-driven constraint-based execution model. International Journal of High Performance Computing Applications 26(3), 319–332 (2012)
3. Frigo, M., Halpern, P., Leiserson, C., Lewin-Berlin, S.: Reducers and other Cilk++ hyperobjects. In: Proceedings of the Twenty-First Annual ACM Symposium on Parallel Algorithms and Architectures, Calgary, Canada (2009)
4. Hewitt, C., Baker, H.G.: Actors and continuous functionals. Technical report, Cambridge, MA, USA (1978)
5. Kleinert, J., Obermayr, M., Balzer, M.: Modeling of Large Scale Granular Systems using the Discrete Element Method and the Non-Smooth Contact Dynamics Method: A Comparison. In: Proceedings of ECCOMAS Multibody Dynamics, Zagreb, Croatia (2013)
6. Robert, J., Halstead, H.: Multilisp: a language for concurrent symbolic computation. ACM Trans. Program. Lang. Syst. 7(4), 501–538 (1985)
7. Sterling, T.: Making slower faster. Presentation at the meeting of the ASCAC Subcommittee on Exascale Research Challenges (August 2013)
8. Tang, M., Manocha, D., Tong, R.: MCCD: Multi-core collision detection between deformable models using front-based decomposition. In: Proceedings of Graphical Models (2010)

Implementation and Optimization of Three-Dimensional UPML-FDTD Algorithm on GPU Clusters

Lei Xu and Ying Xu

Shanghai Supercomputing Center, Shanghai, 201203, China
{lxu,yxu}@ssc.net.cn

Abstract. Co-processors with powerful floating-point operation capability have been used to study the electromagnetic simulations using the Finite Difference Time Domain (FDTD) method. This work focuses on the implementation and optimization of 3D UPML-FDTD parallel algorithm on GPU clusters. A set of techniques are utilized to optimize the FDTD algorithm, such as the application of GPU texture memory, asynchronous synchronization of data transfer between CPU and GPU. The performance of the parallel FDTD algorithm is tested on K20m GPU clusters. The scalability of the algorithm is tested for up to 80 NVIDIA Tesla K20m GPUs with the parallel efficiency up to 95%, and the optimization techniques explored in this study are found to improve the performance.

1 Introduction

Graphic Processing Units (GPU) for general purpose programming have become widely used to accelerate a broad range of applications, including computational physics, image processing, engineering simulations, quantum chemistry, and a lot more [1,11,10]. GPU is rapidly emerging as a cost-efficient platform for high performance parallel computing. The latest GPU, such as the NVIDIA Tesla Kepler K40 computing processor, has 2880 cores and 12 GB fast on-chip memory and supports 1.43 Tflop/s double precision floating point operation and almost 4.28 Tflop/s for the single precision representation.

The Finite-Difference Time-Domain method (FDTD) is a widely used numerical method for the electromagnetic field simulation [6]. The three-dimensional FDTD simulations for real-world applications require a large amount of fast memory and computation time. Recently there are reports about FDTD simulations using GPUs [2,3,5], but most of researches focus on the implementation and optimization on a single GPU card. Kim and Park [3] used the roofline model [7] to analyze the performance of the three-dimensional FDTD on a GPU card. The overlap communication model is also used to optimize the parallel FDTD algorithm on GPU by Kim and Park [2]. The load balancing for FDTD method on heterogeneous GPU clusters has been studied thoroughly by Shams and Sadeghi [5]. These studies show performance results of small scale simulations on less than 10 GPU cards.

In this paper, we will discuss the three-dimensional FDTD code optimized for GPU clusters and use up to 80 NVIDIA Tesla Kepler K20m GPUs for the strong scaling study. The rest of the paper is organized as follows: Section 2 describes the computational model for the simulation code. Section 3 presents the implementation on multi-GPU and the optimization methods. Section 4 discusses the performance results and the concluding remarks are in Section 5.

2 Computational Model for the FDTD Method

The three-dimensional electromagnetic field is governed by the Maxwell's equation:

$$\frac{\partial}{\partial t} \mu(\mathbf{r}) \mathbf{H}(\mathbf{r}, t) = -\nabla \times \mathbf{E}(\mathbf{r}, t) - [\mathbf{M}_{source}(\mathbf{r}, t) + \sigma_m(\mathbf{r}) \mathbf{H}(\mathbf{r}, t)] \quad (1)$$

$$\frac{\partial}{\partial t} \varepsilon(\mathbf{r}) \mathbf{E}(\mathbf{r}, t) = \nabla \times \mathbf{H}(\mathbf{r}, t) - [\mathbf{J}_{source}(\mathbf{r}, t) + \sigma(\mathbf{r}) \mathbf{E}(\mathbf{r}, t)] \quad (2)$$

where ε is the electrical permittivity, σ is the electric conductivity, μ is the magnetic permeability and σ_m is the equivalent magnetic loss. In Eqn. (1), (2), \mathbf{M}_{source} is the equivalent magnetic current density, \mathbf{J}_{source} is the electric current density and these two terms act as independent sources of \mathbf{E} and \mathbf{H} field energy.

Along x , y and z directions, the vector components of curl operators of Eqn. (1), (2) can be written as a system of six coupled scalar equations. For example, the governing equation of H_x can be written as

$$\frac{\partial H_x}{\partial t} = \frac{1}{\mu} \left[\frac{\partial E_y}{\partial z} - \frac{\partial E_z}{\partial y} - (M_{source_x} + \sigma_m H_x) \right] \quad (3)$$

where E_y and E_z denote the electric field in y and z directions. To discretize the six scalar equations in the three-dimensional space, the Yee cell is used [12], where \mathbf{E} components can be identified by the current flux linking \mathbf{H} loops. The leapfrog for the time derivatives is used where \mathbf{E} and \mathbf{H} are separated by $\frac{1}{2}\Delta t$. The second-order center finite-difference scheme is used for the space and time derivatives.

The FDTD approach is usually used to solve the electromagnetic wave interaction problems. The radiated or scattered waves propagate through arbitrary structures in open regions, where the field computation domain is usually unbounded in one or more directions. However, the numerical simulation is limited by the memory and computational power, and the computational domain is truncated to fit in. The boundary conditions at the outer perimeter of the domain suppress spurious reflections of the outgoing numerical waves to an acceptable level, and also ensure the numerical stability of the FDTD solution. The boundary conditions that satisfy the above requirements are usually denoted as absorbing boundary conditions (ABCs), where the mathematical model or an absorbing medium is applied at the outer interface. In this study, the parallel algorithm for the FDTD method with the uniaxial perfect matched layer (UPML) [4] boundary conditions is investigated.

3 Multi-GPU Computations

3.1 Domain Partitioning and Data Communication Patterns

The electromagnetic domain is first decomposed to several sub-domains. From the analysis [9], the domain decomposition in the three-dimensional way is chosen in this study, where the buffer size for communication is found to be smaller compared to 1D and 2D domain decomposition ways.

Since the second order central finite difference is used, the surface electromagnetic data need to be exchanged between the neighboring processors. For the three-dimensional domain decomposition, the data on the subdomain surfaces is transferred with the surrounding six neighboring processors as shown in Fig. 1[9]. For example, $E_y(i, j + 1, k)$ and $E_z(i, j, k + 1)$ are needed in the update of H_x . If the numbers of the grid points along x, y, z directions of the subdomain are denoted as N_x , N_y and N_z , and the MPI processor is indexed as (p, q, r) , then for $j=N_y$, $E_y(i, N_y+1, k)$ resides on MPI processor $(p, q+1, r)$, and for $k=N_z$, $E_z(i, j, N_z+1)$ resides on MPI processor $(p, q, r+1)$, which are all needed to be transferred to the processor (p, q, r) . E_y at the plane $j=1$ and E_z at the plane $k=1$ on the processor (p, q, r) are also required on processor $(p, q-1, r)$ and $(p, q, r-1)$ respectively, and should be sent to these two processors.

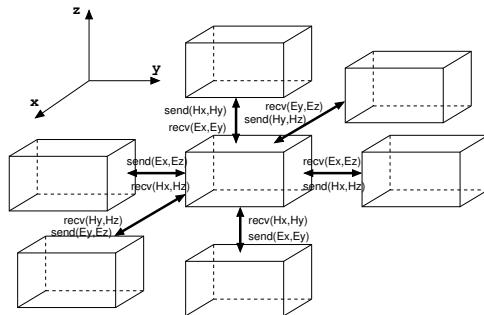


Fig. 1. The electromagnetic field components to be transferred with the six surrounding neighboring processors

3.2 Data Transfer for Multi-GPU Computations

In this study, the number of sub-domain is the same as that of GPU cards used. The electromagnetic field \mathbf{H} and \mathbf{E} are all stored on the global memory of GPUs. However, the data transfer between multi-GPUs is not directly controlled by GPU, so the data communication between multi-GPUs consists of three steps: i) data copy from GPU to host memory, ii) MPI communication, iii) memory copy from host to GPU. The data transfer between CPU and GPU is executed with CUDA APIs.

With the extra overhead of data transfer between host memory and GPU, the main issue affecting the performance of FDTD is the communication overhead, which becomes more critical for GPU computing as the processing capability of a single GPU is found to be much faster than that of CPU cores. The communication overhead could be optimized by overlapping communication and computation. In this section, we presents two communication options: blocking and nonblocking, as shown in Fig. 2. Different from the computational kernel studied in [2,3,5], the computational kernel is the same for the total field and the scatter field with the UPML boundary condition applied. The UPML boundary conditions lays in the coefficients, such as the electrical permittivity, electric conductivity, etc.

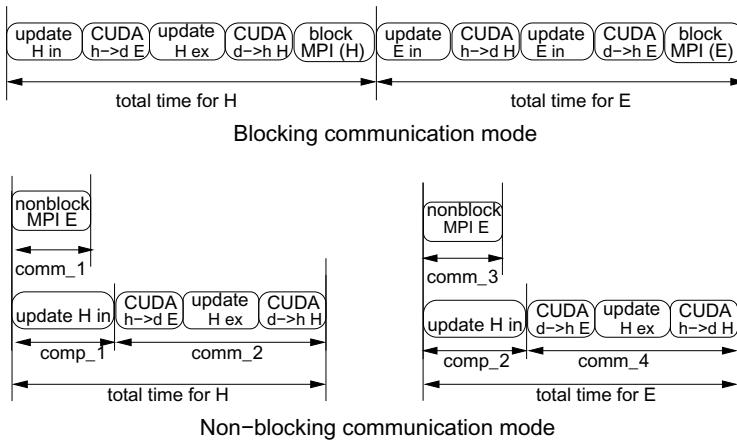


Fig. 2. Multi-GPU computation with MPI blocking and non-blocking communications

For both blocking and non-blocking communication modes, the kernels for calculating the electromagnetic field are divided into two parts: the surface grid points and the internal region. In Fig. 2, “*in*” stands for the internal grid points and “*ex*” denotes the surface grid points. With the blocking communication mode, the internal magnetic field is calculated, the electric data \mathbf{E} required in the update of \mathbf{H} is transferred through CUDA API, then the external layer of \mathbf{H} is calculated. The updated external layer of \mathbf{H} is sent to the neighboring GPUs through CUDA and MPI APIs. The similar operations are applied to the electric field.

From the blocking method, when the data transfer between CPUs is in operation, GPUs are idle, so the update of the internal region and the data communication between CPUs can be done simultaneously. As shown in the non-blocking communication mode in Fig. 2, the non-blocking MPI operation is overlapped with the update of the internal electromagnetic field. Since the data transfer with CUDA API and the update of the boundary grid points are in a sequential

mode and the data received from MPI communication is needed in the update of boundary grid points, these three operations are done in serial after the non-blocking MPI operation.

3.3 Optimization on GPU Kernels

Even with the non-blocking communication used to hide the communication overhead, the performance of GPU kernel can be further improved with some CUDA techniques, such as, the data packing of non-contiguous memory on GPU, the efficient execution of GPU kernels.

Efficient Implementation Using Texture Memory. As described in [8], the texture memory is a read-only memory, resides in the global memory and is cached in two-level cache called texture cache. The cache is optimized for spatial locality in the coordinate system of texture memory. Using the texture memory for GPU computing could enable fast random or non-aligned access to arrays and the texture cache could also provide bandwidth aggregation. The texture memory as a read path could work around the coalescing constraints, reduce the external bandwidth requirements, or both[8].

In the FDTD method, the coefficients in Eq. 3 only vary one direction and are stored in one-dimensional arrays. In the GPU computational kernel, these 1D coefficient arrays are bind with the texture memory to accelerate the reading of coefficients.

Asynchronous Mode for Boundary Data Packing and Transferring. As shown in Section 3.2, after the update of external points, the boundary layer of the domain is sent from GPU to CPU. Each time two surfaces are packed and sent to the host memory, then the packed data is transferred using MPI APIs. The boundary data is not continuous in the memory and data packing is required.

The data packing and transferring can be done in a sequential mode. In the “sync mode” of Fig. 3, the “right” and “top” surfaces are first packed and then sent to the host memory with CUDA API “cudaMemcpy”. The data of the two surfaces is sent using one blocking MPI routine.

In the asynchronous mode of Fig. 3, cudaStream is created for the two boundary surfaces, and the boundary data is packed and transferred to the pinned memory on the host memory through “cudaMemcpyAsync” simultaneously and then the data is sent with two MPI non-blocking routines. The asynchronous mode takes advantage of the GPU computational power and the PCI-E bandwidth and is expected to increase the efficiency of GPU kernels .

The Concurrent GPU Kernels Execution. If only one kernel function is used to calculate the electromagnetic field update, the logic branching is required to process the position of boundary grid points, which is expected to significantly

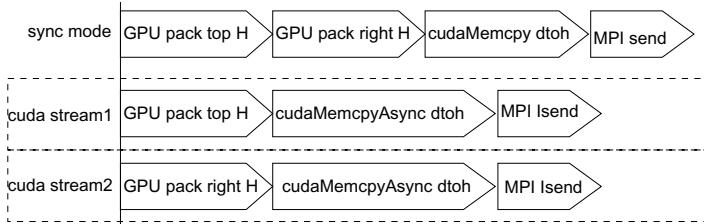


Fig. 3. Boundary data pack and transfer in synchronous and asynchronous modes

lower the efficiency of the thread execution. This is because if one thread is used to calculate one grid point, the number of threads in the parallel execution for the surface grid points is very small and most of the GPU computation capability will be idle. NVIDIA GPU hardware with compute capability 2.X supports the concurrent kernel execution, where different kernels of the same application work together, which is expected to increase the efficiency of GPU. For the FDTD algorithm, the computation of boundary grid points can be done with multiple kernels simultaneously and hence increases the execution efficiency on GPU.

4 Results

This section presents the test results and the performance gain obtained by the optimization techniques. The GPU implementation of FDTD algorithm is tested on the π cluster installed at Shanghai Jiaotong University. The π cluster has 50 compute nodes integrated with 100 NVIDIA Tesla K20m GPU cards. The compute nodes are connected through the Mellanox Infiniband FDR switch. For all tests, CUDA 5.0, Intel icpc 13.1.1 compiler, and Intel MPI 4.1.1 library are used. The performance of two problem sizes $N = 448^3, 768^3$ is plotted in Fig. 4 in terms of Tflop/s.

4.1 Parallel Efficiency Analysis

The CPU version with the non-blocking communication has performance as 1/15 of that of the GPU software, but the parallel efficiency of the CPU version reaches 97.1% for $N_p=64$ for the problem size 448^3 . The almost linear speedup observed in the CPU software is due to the small amount of communication data transferred in the FDTD parallel algorithm and also the high-speed FDR Infiniband switch on π cluster.

For the GPU implementation, the performance of optimized GPU kernel is compared with that of the GPU kernels without optimization techniques discussed in Section 3.3 for $N = 448^3$. The performance gain due to the optimized techniques is more than 10%, and the scalability of the parallel GPU software is also improved, where the parallel efficiency increases from 71% to 84% for 64 GPU cards. It is noted that the optimized GPU code has parallel efficiency less

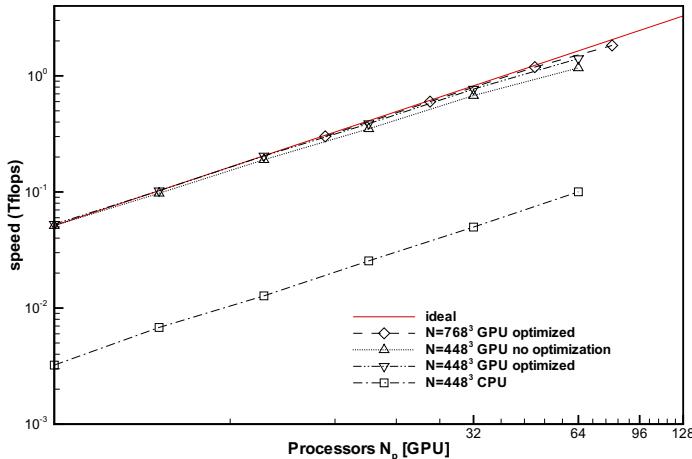


Fig. 4. The strong scaling for two problem size $N = 448^3$ and $N = 768^3$ on π cluster

than that of pure CPU implementation and this is because of the communication overhead between host memory and GPU.

For a certain problem size, as the number of GPU cards increases, the parallel efficiency decreases. This is because the computational time is determined by the total number of grid points of the sub-domain which is of order $\mathcal{O}(N^3)$, while the communication time is related to the surface grid points of the sub-domain as the order of $\mathcal{O}(N^2)$. With the problem size decreasing, the decrease of computational time is expected to be faster than that of communication time. It is observed that the optimization techniques described in Section 3 improves the parallel efficiency, but as the number of GPU cards increases, the performance gain from the overlapping method decreases.

For the larger problem size $N = 768^3$, the total memory required is 42.8 GB. It is observed that the optimized GPU code shows the linear scalability with parallel efficiency higher than 95% for 12, 24, 48 and 80 GPUs. As the problem size increases, the GPU computational time is large enough to cover the communication overhead, and hence parallel efficiency for the larger problem size is found to be almost linear for 80 GPU cards.

4.2 Performance Analysis Using Roofline Model

It is worth noting that our study has performance of 26 Gflops/s or 0.36 Gpoints/s per GPU card and is less than performance result reported in [3,2] which calculates 1 Gpoints/s per GPU on NVIDIA Tesla C2050. However these studies investigated the Maxwell's equation without boundary conditions and the floating point operation per grid point is 30 and the operational intensity is 0.3125 flop/Byte according to the roofline model.

The FDTD method with UPML boundary conditions in this work involves solving additional equations as discussed in Section 2. The number of floating point operation in this study is 78 and the operational intensity is 0.1625, as the coefficients such as the electrical permittivity, the electric conductivity and etc. vary with locations. The attainable performance from NVIDIA K20m is 33.8 Gflop/s for the maximum bandwidth 208GB/s and 22.8 Gflop/s for the measured bandwidth 140.5GB/s. It is observed that the floating point operation per GPU card for the optimized GPU software is 23 ~ 25 Gflop/s which is 10% higher than that predicted by the roofline model. This indicates that the optimization techniques discussed in Section 3.2 and 3.3 improved the performance.

5 Summary and Discussion

The three-dimensional FDTD simulation for multiple GPU cards is discussed in this paper, where GPUs conduct all the computation of the electromagnetic field and the inter-processor communication is implemented with MPI. To optimize the GPU software, the communication overhead is reduced by splitting the computational kernels on GPUs and overlapping the MPI communication with the computation. The GPU implementation is tested with up to 80 NVIDIA Tesla K20m GPUs. The optimization techniques investigated in this study are found to increase the parallel efficiency of the GPU software. For the problem size 448^3 , the parallel efficiency is improved from 71% to 84% for 64 GPU cards. For the larger problem size 768^3 , the parallel efficiency is 95% for 80 GPU cards and the GPU implementation achieves a speed of 26 Gflops/s per GPU card.

In this study, all the computation is executed solely on GPUs and the electromagnetic field is stored in the global memory of GPU. In this way, the computational capability of GPU is fully utilized, but the FDTD problem size solved with this GPU software is limited by the global memory on GPUs. One way to work around this is to use the memory and computation cores of CPU. The collaborative computing of GPU and CPU becomes more urgent as the computation capacity of CPU increases dramatically and the priority of the HPC system's energy consumption becomes higher. This will be investigated in the future work.

Acknowledgment. This work is financially sponsored by the National High Technology Research and Development Program (863 project) under the grant No. 2012AA01A308. The computational resource is supported by the Center for High Performance Computing of Shanghai Jiaotong University and by the Shared Hierarchical Academic Research Computing Network and Compute/Calcul Canada.

References

1. Egri, G.I., Fodor, Z., Hoelbling, C., Katz, S.D., Ngrdi, D., Szab, K.K.: Lattice QCD as a video game. *Computer Physics Communications* 177(8), 631–639 (2007)
2. Kim, K.-H., Park, Q.-H.: Overlapping computation and communication of three-dimensional FDTD on a GPU cluster. *Computer Physics Communications* 183, 2364–2369 (2012)
3. Kim, K.H., Kim, K.-H., Park, Q.-H.: Performance analysis and optimization of three-dimensional FDTD on GPU using roofline model. *Computer Physics Communications* 182, 1201–1207 (2011)
4. Sacks, Z.S., Kingsland, D.M., Lee, R., Lee, J.F.: A perfectly matched anisotropic absorber for use as an absorbing boundary condition. *IEEE Trans. Antennas and Propagation* 43, 1460–1463 (1995)
5. Shams, R., Sadeghi, P.: On optimization of finite-difference time-domain (FDTD) computation on heterogeneous and GPU clusters. *J. Parallel Distrib. Comput.* 71
6. Taflove, A., Hagness, S.C.: Computational Electrodynamics: The Finite-Difference Time-Domain Method. The Artech House antenna and propagation library. Artech House, Incorporated (2005)
7. Williams, S., Waterman, A., Patterson, D.: Roofline: An insightful visual performance model for multicore architecture. *Communications of the ACM* 52(4), 65–76 (2009)
8. Wilt, N.: CUDA Handbook: A Comprehensive Guide to GPU Programming. Addison-Wesley Professional (2013)
9. Xu, L., Xu, Y., Jiang, R.L., Zhang, D.D.: Implementation and optimization of three-dimensional UPML-FDTD algorithm on GPU cluster. *Computer Engineering & Science* 35(11), 29–36 (2013)
10. Yang, J., Wang, Y., Chen, Y.G.: accelerated molecular dynamics simulation of thermal conductivities. *Journal of Computational Physics* 221(2), 799–804 (2007)
11. Yasuda, K.: Two-electron integral evaluation on the graphics processor unit. *Journal of Computational Chemistry* 29(3), 334–342 (2008)
12. Yee, K.: Numerical solution of initial boundary problems involving maxwell's equations in isotropic media. *IEEE Transactions on Antennas and Propagation* (14), 302–307 (1966)

Real-Time Olivary Neuron Simulations on Dataflow Computing Machines

Georgios Smaragdos¹, Craig Davies³, Christos Strydis¹, Ioannis Sourdis⁵,
Cătălin Ciobanu⁵, Oskar Mencer^{3,4}, and Chris I. De Zeeuw^{1,2}

¹ Dept. of Neuroscience, Erasmus Medical Center, Rotterdam, The Netherlands

² Netherlands Institute for Neuroscience, Amsterdam, The Netherlands

³ Maxeler Technologies Inc., UK

⁴ Imperial College London, UK

⁵ Dept. of Computer Science & Engineering, Chalmers University of Technology,
Gothenburg, Sweden

g.smaragdos@erasmusmc.nl

Abstract. The Inferior-Olivary nucleus (ION) is a well-charted brain region, heavily associated with the sensorimotor control of the body. It comprises neural cells with unique properties which facilitate sensory processing and motor-learning skills. Simulations of such neurons become rapidly intractable when biophysically plausible models and meaningful network sizes (at least in the order of some hundreds of cells) are modeled. To overcome this problem, we accelerate a highly detailed ION network model using a Maxeler Dataflow Computing Machine. The design simulates a 330-cell network at real-time speed and achieves maximum throughputs of 24.7 GFLOPS. The Maxeler machine, integrating a Virtex-6 FPGA, yields speedups of $\times 92\text{--}102$, and $\times 2\text{--}8$ compared to a reference-C implementation, running on a Intel Xeon 2.66GHz, and a pure Virtex-7 FPGA implementation, respectively.

1 Introduction

The United-States National Academy of Engineers has classified brain simulation as one of the Grand Engineering Challenges [7]. Biologically accurate brain simulation is a highly relevant topic for *neuroscience* for a number of reasons; the main goal is accelerated brain research by the creation of more advance research platforms. Even though a significant amount of effort is spent in understanding brain functionality, the exact mechanisms in most cases are still only hypothesized. Fast and real-time simulation platforms can enable the the neuroscientific community to more efficiently test these hypotheses. Better understanding of brain functionality can lead to a number of critical practical applications: (1) *Brain rescue*: If brain functions can be simulated accurately enough and in real-time, this can lead to robotic prosthetics and implants for restoring lost brain functionality. (2) *Advanced A.I.*: Artificial Neural Networks (ANNs) have already been successfully used in this field. It is believed that greater understanding of biological systems and the richer computational dynamics of their

models, like *spiking neural networks* (SNNs) [5], can lead to more advanced, artificial-intelligence and robotic applications. (3) *New architectural paradigms: Alternatives to the typical Von-Neumann architectures.*

In many of these applications, real-time performance of Neural Networks (NN) is desirable or, even, required. The main challenge in building such systems lies largely in the computational and communication load of the network simulations. Furthermore, biological NNs execute these computations with massive parallelism, something that conventional, CPU-based execution cannot cope with very well. As a result, the neuron-population size and interconnectivity are quite low when running on CPUs. This greatly impedes the efficiency of brain research in relation to the above goals of brain simulation.

Other HPC computing alternatives fall short in a number of aspects. *General-Purpose GPUs* can exploit application parallelism better and can be more efficient in running neuron models. Yet, in the case of complex models or very large-scale networks, they may not be able to provide real-time performance due to the high rates of data exchange between the neurons and are less efficient in terms of energy and power. *Supercomputers* can emulate the behavior and parallelism of biological networks with sufficient speed but have limited availability and require large space, implementation, maintenance and energy costs while lacking any kind of mobility. *Mixed-signal, Very-Large-Scale-Integration (VLSI) circuits* achieve adequate simulation speeds while simulating biological systems more accurately since they model neurons through analog signals. However, they are much more difficult to implement, lack flexibility and often suffer from problems typical in analog design (transistor mismatching, crosstalk etc.).

Implementing the neural network in parallel, digital hardware can efficiently match the parallelism of biological models and provide real-time performance. While *Application-Specific Integrated-Circuits* (ASICs) design is certainly an option, it is expensive, time-consuming and – most importantly – inflexible (i.e. cannot be altered after fabrication), a characteristic often required when *fitting* novel neuron models. Most of these issues can be tackled through the use of reconfigurable hardware. Although slower than ASICs, it can still provide high performance for real-time and hyper-real-time neuron simulations, by exploiting the inherent parallelism of hardware. Besides requiring a lot less energy and, in some cases, less space than most of the above solutions for delivering the same computational power, the reconfiguration property of such platforms provides the flexibility of modifying brain models on demand.

In this paper we present a hardware-accelerated application for a specific, biophysically-meaningful SNN model, using single-precision floating-point (FP) arithmetic computations. The application is mapped onto a Maxeler Datalow Machine [9] which employs a dataflow-computing paradigm utilizing highly efficient reconfigurable-hardware-based engines. The targeted application models an essential area of the Olivocerebellar system: the Inferior-Olivary Nucleus (ION). This design is a part of an ongoing effort to replicate the full Olivocerebellar circuit on reconfigurable hardware-based platforms, for the purpose of real-time experimentation on meaningful network sizes.

2 The Biological Neuron, the Cerebellum and the Inferior Olive

Neurons are electrochemically excitable cells¹ that process and transmit signals in the brain. The biological neuron comprises in general (although, in truth is a much more complicated system) three parts, called compartments in neuromodelling jargon: The *Dendrites*, the *Soma* and the *Axon*. The dendritic compartment represents the cell input stage. In turn, the soma processes the stimuli and translates them into a cell *membrane potential* which evokes a cell response called an *action potential* or, simply, a *spike*. This response is transferred through the axon – the cell output stage – to other cells through a *synapse*.

The cerebellum is one of the most complex and dense regions in the brain, playing an important role in sensorimotor control. It does not initiate movement but influences the sensorimotor region in order to precisely coordinate the body's activities and motor learning skills. It also plays an important role in the sensing of rhythm, enabling the handling of concepts such as music or harmony. The Olivocerebellar circuitry is an essential part of the Cerebellar functionality. Main areas of the circuit are the *Purkinje cells* (PC), the *Deep Cerebellar Nuclei* (DCN) and the *Inferior-Olivary Nucleus* (ION) [3]. The system is theorized to be organized in separate groups of PC, DCN and ION cells (*Microsections*) that – in combination – control distinctive parts of muscle movement and may or may not be interconnected to each other [8]. The network sizes in these microsections are hypothesized to vary. Specifically for the ION cell, their numbers in a microsection can be from a dozen to several hundreds of cells. ION cells especially are also interconnected by purely electrical connections between their dendrites, called *gap junctions*, considered to be important for the synchronization of activity within the nucleus and the Olivocerebellar system, in general.

3 Related Work

In the past, a number of designs have been proposed for the implementation of neuron and network models using reconfigurable hardware. In this section, we present some of the most notable works in the field.

Two of the most interesting implementations in terms of *computation (performance) density* [10] using Izhikevich neuron modeling [4] are the designs proposed by Cheung et al. [2] and by Moore et al. (Bluehive [6]). Each approach proposed a reconfigurable-hardware architecture for very-large-scale SNNs. To improve on their initial memory-bandwidth requirements, Cheung et al. also used a Maxeler Dataflow Machine [2]. The size of the implemented network achieved was 64K neurons, each having about 1,000 synaptic connections to neighboring neurons. In the Bluehive device, Moore et al. implemented DDR2 RAMs and built custom-made SATA-to-PCI connections for stacking FPGA devices for facilitating large SNN simulations. Only a small portion of data was

¹ We will use the terms *neuron* and *(neural) cell* interchangeably in this paper.

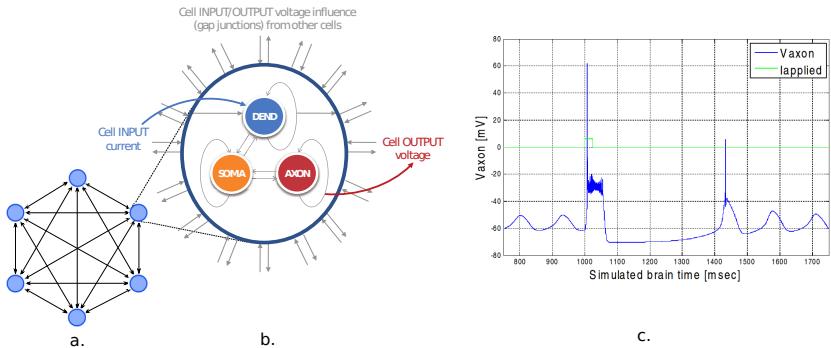


Fig. 1. Illustration of (a) a 6-cell network-model example, (b) the 3-compartmental model of a single ION cell, and (c) its output (spiking) response to an input pulse

stored on-board the FPGAs. In a Stratix IV FPGA, the authors simulated 64K Izhikevich neurons with 64M simple synapses at real-time performance.

These works have incorporated fixed-point arithmetic to implement the computation of their neuron models. Zhang et al. [11] have proposed a Hodgkin-Huxley (HH) modeling [4] accelerator using dedicated Floating Point (FP) units. The FP units were custom-designed to provide better accuracy, resource usage and performance. The 32-bit FP arithmetic used in the model produced a neuro-processor architecture which could simulate 4 complete cells (synapse, dendrite, soma) at real-time speed. Smaragdos et al. [10] have also ported an HH-based model of Olivocerebellar cells onto a Xilinx Virtex-7 FPGA. The model entails demanding FP computations per cell as well as high inter-cell communication (up to all-to-all connectivity). Real-time simulations of up to 96 neurons could be achieved. It must be noted that the neuron model used was estimated about x18 more complex compared to the rest of the related works in terms of FP operations.

4 ION-Model Description

The application accelerated in this paper models the behavior of the ION neurons using an extended-HH model description [1]. This model is a first step towards building a high-performance, Olivocerebellar-system simulation platform. The model not only divides the cells into multiple compartments but also creates a network onto which neurons are interconnected by modeling gap junctions.

The ION cell model divides the neuron into three computational compartments – closely resembling biological counterparts – as shown in Figure 1(b). Every compartment has a state that holds pertinent electrochemical variables and, on every simulation step, the state is updated based on: i) the previous state, ii) the other compartments' previous state, iii) the other neurons' previous state, and iv) any externally evoked input stimulus to the cell or network.

The computational model operates in a fashion allowing *concurrent* execution of the three compartments. The model is calibrated to produce every per-neuron output value with a 50 μ sec time step. This means that, in order to support real-time simulations, all neurons are required to compute one iteration of compartmental calculations within 50 μ sec. Due to the realistic electrochemical variables handled by the model, most of the computations require FP arithmetic.

Figure 1(a) illustrates the network-model architecture with an example size of 6 cells. Every cell receives, through the dendritic compartment, the influence of all other cells in the network, thus modeling the massive number of biological gap junctions present in the Inferior Olive. The gap-junction computations are repeated for every neighboring input, computing the accumulated influence of all neighboring connections. The ION network must be synchronized in order to guarantee the correct exchange of cell-state data when multiple cells and compartments are being computed simultaneously. The system works in lock-step, computing discrete output values (with a 50- μ sec time step) that, when aggregated in time, contribute to form the electrical waveform response of the system (Figure 1(c)).

An extensive profiling of the application, conducted in [10], reveals that in a fully interconnected ION network the gap-junction computations increase quadratically with network size. A meaningful network size for the ION model would be one large enough to enable as extensive as possible exploration of microsection organizations. Organizations of either a high number of microsections with few cells or a low number of microsections with many cells can reveal a range of different system behavioral dynamics and, thus, validate (or invalidate) a set of hypotheses on brain functionality. According to our neuroscientific expert partners, a minimum ION-network size for meaningful experimentation would be around 100 cells. Any improvement in achievable size beyond this point would enable greater possibilities for the exploration of microsection behavior. For such network sizes (100 cells and higher), gap-junction computations dominate all cell computations.

5 DFE Implementation of the ION Model

The design presented in this paper is a continuation of a previous work of an FPGA implementation of an ION-network hardware accelerator [10]. The FPGA kernel was designed to work alongside a host processor and executed the aforementioned model in a step-by-step process. The hardware description of the kernel was coded in C using the Vivado HLS v2013.2 tool targeting a Virtex-7 device. Although this design offered considerable speedup over a reference CPU implementation, it was still unable to fully exploit the parallelism of the model – which essentially is a dataflow application – without considerable restructuring of the initial model.

On the other hand, a Maxeler Dataflow Computing Machine [9], based on *data-flow engines* (or DFEs), has the ability to better exploit the inherent parallelism of the model and has the potential to achieve even greater speed-ups

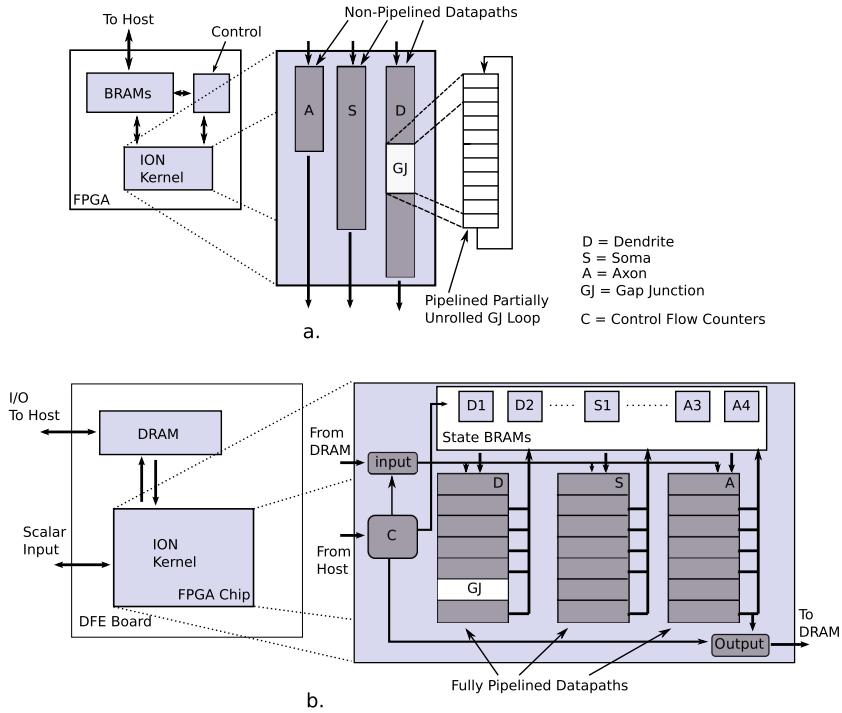


Fig. 2. Illustration of (a) A single instance of the FPGA ION Kernel of [10] (b) A single instance of the DFE ION Kernel

with minor changes in the model architecture. Also based on FPGA hardware, DFE boards and their compiling tools are designed to fully exploit loop-level parallelism by fine-grain pipelining of computations and efficient usage of the FPGA fabric, while additionally including efficient I/O between the chip and the on-board resources (e.g. on-board DRAMs).

5.1 The ION-Kernel DFE Architecture

The DFE implementation of the ION network can be seen in Figure 2(b). The design incorporates 3 internal pipelines one for each part of the cell (Dendrite, Soma, Axon), executing the respective computations. The cell states consist of 19 FP values. Each parameter for each neuron is stored on its own BRAM block, for fast read/update of the network state. Since every new cell state is dependent only on the network state of the previous simulation step, only one copy of each neuron state is required. The input stream of the DFE kernel comes from the on-board RAM and represents the evoked inputs (one value for each neuron per simulation step) used in the dendritic computations comprising the network input. Only for the first simulation step the initial state and neighboring (gap-junction) influence are also streamed-in from the on-board memory as each

neuron begins its first simulation step. The network output (represented by the axonal voltage) is also streamed to the on-board memory at the same point as it is updated on its respective BRAM block.

Due to the dataflow paradigm followed, the DFE kernel executes the complete simulation run when activated, as opposed to the control-flow-based FPGA kernel that only executes the simulation step by step, under the supervision of a MicroBlaze core. As such, the DFE kernel additionally receives scalar input parameters, denoting the simulation duration and the network size to be simulated. Program flow is monitored using hardware counters monitoring gap-junction loop iterations, neurons executed and the number of simulation steps concluded. All scalar parameters, activation of the kernel, input-data preparation before execution and output visualization after execution is handled by an off-board host processor. The data flows through the DFE pipelines with each kernel execution step (or *tick*), consuming the respective input or producing the respective output and state. Each kernel tick represents the completion of one gap-junction loop iteration. As a result, the DFE execution naturally pipelines not only the gap-junction loop iterations but the execution of different neurons within one simulation step as well, as opposed to the FPGA kernel that was capable of only the former (Figure 2(a)). Simulation steps are not pipelined in an all-to-all network, as every neuron must have the previous state of all other neurons ready for its gap-junction computations before a new step begins. The DFE pipeline is, thus, flushed before a new simulation step begins execution. The execution of a single simulation step requires N^2 ticks to be completed, where N is the network size.

5.2 Additional Design Optimizations

There are two straightforward ways to speed up execution of the DFE kernel. One is to use multiple instances of the kernel in a single DFE, if the DFE spare resources allow it, thus doubling the network size achievable within a certain time frame. The other is to unroll the gap-junction loop by replicating the single-iteration hardware logic, essentially executing multiple iterations of the loop per kernel tick. If U is the unroll factor of the loop, the number of ticks required for a network simulation step is $N * N/U$, denoting potentially a considerable speed-up. Both of these techniques are subject to area but also timing constraints. Loop unrolling, in particular, could cause extra pressure in the routing of the hardware, limiting the maximum achievable frequency of the DFE kernel.

6 Evaluation

The design was implemented on a Maxeler device using the Vectis architecture. The Vectis boards include a Virtex 6 SX475T FPGA in their DFEs. The maximum frequency that an ION kernel achieved on the DFE board was 150MHz. This design could be optimized by either using a second kernel instance within the same DFE or unrolling the gap-junction loop. Unfortunately, the spare resources did not enable the use of both optimizations simultaneously. As a result,

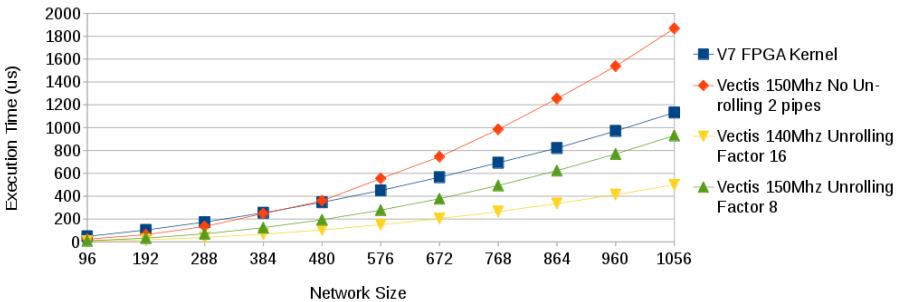


Fig. 3. Simulation Step execution time for the DFE kernels and the FPGA kernel [10].

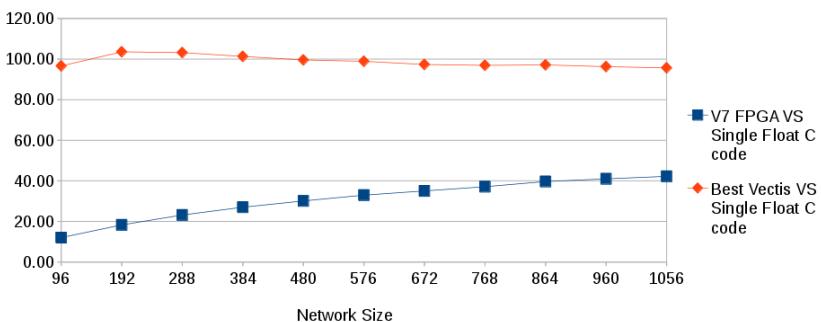


Fig. 4. Speed-up of best DFE and FPGA implementations compared to single-FP CPU-based execution on an Intel Xeon 2.66GHz with 20GB RAM

2 versions of the design were tested, one with 2 ION kernels within the DFE and one with a single kernel and loop unrolling. The maximum unroll factor achieved for the frequency of 150MHz was 8. One last design was also evaluated. By reducing the DFE frequency to 140 MHz, the unroll factor could be raised to 16 expecting to balance out the performance loss due to the lower frequency. Larger unroll factors were not achievable due both to timing and area constraints.

In Figure 3 we can see the execution time for all the DFE-based designs and the FPGA-kernel version (deployed on a Virtex 7 XC7VX485T device running at 100 MHz²) which includes 8 instances of the ION kernel shown in Figure 2(a). Indeed, the best-performing Vectis implementation is the one with the lowest frequency but the highest unroll factor. The gain of loop unrolling supersedes the gains of using the extra kernel instance or the higher frequency. The FPGA implementation incorporates also unrolling optimizations but of lower factor (4) and, combined with its coarser-grain pipelining and lower operating frequency, performs worse than the DFE implementation. In effect, the DFE can simulate

² For fairness in comparisons, the Maxeler DFE and the Xilinx FPGA board contain similar resources.

Table 1. Overview of current and related work SNN Implementations on achievable real-time network sizes. CPU Speed-up for the ION designs is compared to a Xeon E5430 2.66GHz/20GB RAM.

Design	Cheung et al. [2]	Smaragdos et al. [10]	This work
Model	Izhikevich	Extended HH	Extended HH
Time Step (ms)	1	0.05	0.05
Real-Time Network Size	64000	96	330
Arithmetic Precision	Fixed Point	Floating Point	Floating Point
Neuron Model OPs * Net. Size (MFLOPS)	> 832*	2131.2	24684
Speed-up vs. CPU	-	x12.5 (C Code)	x92 - x102 (C Code)
FPGA Chip	Virtex 6 SX475T Maxeler Machine	Virtex 7 XC7VX485T	Virtex 6 SX475T Maxeler Machine
Device capacity (LUTs)	297,600 6-input LUTs	303,600 6-input LUTs	297,600 6-input LUTs
Computation density (FLOPS/LUT)	2,796*	7,019	82,943

* Fixed-point operations

up to 330 Inferior-olivary cells at real-time speed (within the 50- μ sec deadline) and is $\times 7.7$ to $\times 2.26$ faster than the FPGA implementation. In terms of speed-up compared to single-core CPU execution³, the fastest DFE implementation has a speed-up of $\times 92$ to $\times 102$ compared to the single-FP C implementation (Figure 4). It uses about 74% of the total logic available in the DFE hardware; more specifically, about 64% of LUTs, 60% of FF, 30% of DSPs and 41% of on-chip BRAMs. The maximum network size that can be simulated is 7,680 ION neurons before we run out of resources.

To quantify the computation density of the design, we use the same method of calculating FP operations per second (FLOPS) per logic unit (LUTs) as in [10]. To the best of our knowledge, the only other SNN implementation on a Maxeler DFE is the one by Cheung et al. [2]. A comparison of this work to the FPGA-based kernel and our new Maxeler-based design can be seen on Table 1. The Maxeler-based design can achieve 24.7 GFLOPS when executing its maximum real-time network (330 cells) and has a computation (performance) density of 82,943 FLOPS/LUT (6-input LUTs), as opposed to 2.1 GFLOPS and 7,019 FLOPS/LUT for the conventional FPGA kernel, respectively.

³ We use the single-core C implementation as a reference point. It would be possible and interesting explore a multi-core implementation and assess the speedups, but this subject is out of the scope of this paper.

7 Conclusions

In this paper a dataflow implementation of a model of the Inferior-olivary Nucleus was presented with significant speed-ups compared to the CPU implementation of the same model and related works. The model itself is a highly complex, extended-HH, biophysically-meaningful model of its biological counterpart. The inherent application parallelism was exploited to a much greater extent when implemented in a single DFE of a Maxeler Dataflow Machine. This, alongside with the higher operating frequency, led to a significant improvement to the previous design implemented on a conventional FPGA board. The fastest DFE implementation achieved a real-time network of 330 neurons – $\times 3.4$ larger than the FPGA one – and achieved almost x2-x8 greater speed-ups compared to the FPGA port. The real-time network supports little over than 24 GFLOPS and has almost x11 greater computation density than the conventional FPGA.

The larger real-time-network size as well as the high modeling accuracy, have the potential to enable deeper exploration of the Olivocerebellar microsection behavior compared to the previous FPGA implementation. Besides the speedup, this DFE-based implementation has also opened new possibilities for future simulation-based brain research: The Maxeler DFE platforms offer extended capabilities and significantly higher programming ease compared to conventional FPGAs. Such capabilities, include the use of the large DRAMs located on the DFE boards, fast network connections directly to the DFE fabric and the ability to combine multiple DFE boards for running massive-scale network simulations.

References

1. Bazzigaluppi, P., De Gruijl, J.R., Van Der Giessen, R.S., Khosrovani, S., De Zeeuw, C.I., De Jeu, M.T.G.: Olivary subthreshold oscillations and burst activity revisited. *Frontiers in Neural Circuits* 6(91) (2012)
2. Cheung, K., Schultz, S.R., Luk, W.: A large-scale spiking neural network accelerator for FPGA systems. In: Villa, A.E.P., Duch, W., Érdi, P., Masulli, F., Palm, G. (eds.) ICANN 2012, Part I. LNCS, vol. 7552, pp. 113–120. Springer, Heidelberg (2012)
3. De Zeeuw, C.I., Hoebeek, F.E., Bosman, L.W.J., Schonewille, M., Witter, L., Koekkoek, S.K.: Spatiotemporal firing patterns in the cerebellum. *Nat. Rev. Neurosci.* 12(6), 327–344 (2011)
4. Izhikevich, E.: Which Model to Use for Cortical Spiking Neurons? *IEEE Trans. on Neural Net.* 15(5) (2004)
5. Maass, W.: Noisy Spiking Neurons with Temporal Coding have more Computational Power than Sigmoidal Neurons. *Neural Inf. Proc. Systems*, 211–217 (1996)
6. Moore, S.W., Fox, P.J., Marsh, S.J., Markettos, A.T., Mujumdar, A.: Bluehive — A Field-Programmable Custom Computing Machine for Extreme-Scale Real-Time Neural Network Simulation. In: IEEE Int. Symp. on FCCM, pp. 133–140 (2012)
7. National Academy of Engineering. Grand Challenges for Engineering (2010)
8. Marshall, S.P., Lang, E.J.: Inferior Olive Oscillations Gate Transmission of Motor Cortical Activity to the Cerebellum. *The Journal of Neuroscience* 24(50), 11356–11367 (2004)

9. Maxeler Technologies. MPC-X Series,
<http://www.maxeler.com/products/mpc-xseries/>
10. Smaragdos, G., Isaza, S., Eijk, M.V., Sourdis, I., Strydis, C.: FPGA-based Biophysically-Meaningful Modeling of Olivocerebellar Neurons. In: 22nd ACM/SIGDA Int. Symposium on FPGAs (FPGA) (2014)
11. Zhang, Y., McGeehan, J.P., Regan, E.M., Kelly, S., Nunez-Yanez, J.L.: Biophysically Accurate Floating Point Neuroprocessors for Reconfigurable Logic. IEEE Trans. on Computers 62(3), 599–608 (2013)

Tofu Interconnect 2: System-on-Chip Integration of High-Performance Interconnect

Yuichiro Ajima, Tomohiro Inoue, Shinya Hiramoto, Shunji Uno, Shinji Sumimoto,
Kenichi Miura, Naoyuki Shida, Takahiro Kawashima, Takayuki Okamoto,
Osamu Moriyama, Yoshiro Ikeda, Takekazu Tabata, Takahide Yoshikawa,
Ken Seki, and Toshiyuki Shimizu

Fujitsu Limited, Kawasaki, Japan

{aji, inoue.tomohiro, hiramoto.shinya, s-uno, sumimoto.shinji,
k.miura, shidax, t-kawashima, tokamoto, moriyama.osamu,
ikeda.yoshir-02, tabata.takekazu, yoshikawa.takah,
seki.ken, t.shimizu}@jp.fujitsu.com

Abstract. The Tofu Interconnect 2 (Tofu2) is a system interconnect designed for the Fujitsu's next generation successor to the PRIMEHPC FX10 supercomputer. Tofu2 inherited the 6-dimensional mesh/torus network topology from its predecessor, and it increases the link throughput by two and half times. It is integrated into a newly developed SPARC64TM processor chip and takes advantages of system-on-chip implementation by removing off-chip I/O between a processor chip and an interconnect controller. Tofu2 also introduces new features such as the atomic read-modify-write communication functions, the session-mode control queue for the offloading of collective communications, and harmless cache injection technique to reduce communication latency.

Keywords: system-on-chip, many-core, supercomputer, interconnect, architecture, RDMA, atomic operations, cache injection, strong atomicity, non-blocking collective communication.

1 Introduction

The Tofu Interconnect 2 (Tofu2) is a system interconnect designed for the Fujitsu's new generation supercomputer which is provisionally called Post-FX10 and will start shipping in 2015. The Fujitsu's PRIMEHPC FX series includes highly scalable, scalar-processor-based supercomputers, and its system-on-chip (SoC) design has been continuously enhanced. The SoC design is necessary for the exascale era because it improves packaging density, system scalability, and energy efficiency by reducing the number of components and off-chip I/Os. PRIMEHPC FX10 [1], which is the previous generation, has a memory controller integrated into a processor chip. In Post-FX10, a newly developed processor chip SPARC64TM XIfx integrates an interconnect controller (ICC).

Tofu2 is the successor of the Tofu interconnect [2,3,4] of the K computer [5] and PRIMEHPC FX10 and inherits the highly scalable and fault-tolerant 6-dimensional

(6D) network that successfully built the K computer by interconnecting 88,128 processors. In this paper, the Tofu interconnect is called Tofu1. Tofu stands for “torus fusion” or “torus-connected full-connection”.

Tofu2 increases the link throughput by two and half times. The enhanced bandwidth of Tofu2 feeds data to the SPARC64TM XIfx processor, which has 32 compute cores and delivers over 1-TFlops double-precision and over 2-TFlops single-precision floating-point performance. The SPARC64TM XIfx processor also has two additional cores called assistant cores for use in system services and communication assistance. Basically, Tofu2’s user-mode functions are intended to be used from compute cores and the system-mode functions are designed for assistant cores. An assistant core may use the user-mode functions of Tofu2 in cases in which that a communication library creates a client thread to process inter-process protocols in the background.

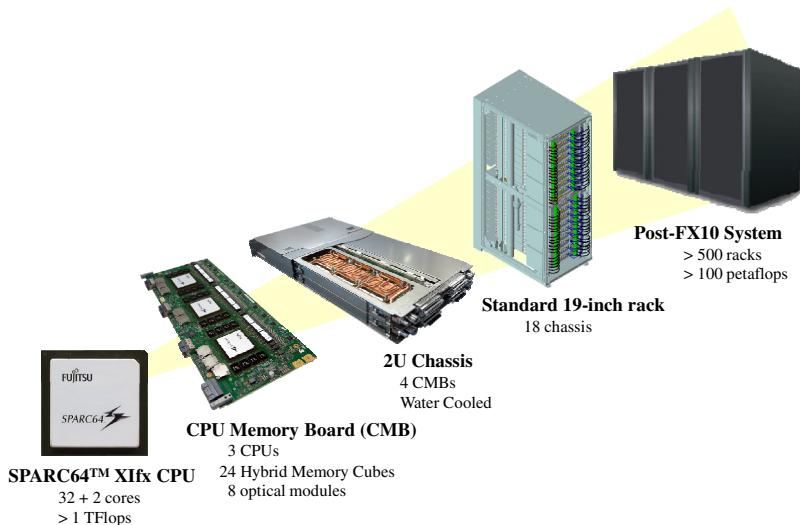


Fig. 1. Post-FX10 packaging hierarchy

Fig. 1 shows the packaging hierarchy of Post-FX10. A Post-FX10 node consists of a SPARC64TM XIfx processor chip with eight high-speed Hybrid Memory Cubes. A CPU memory board (CMB) loads three nodes and each chassis houses four CMBs. A total of twelve nodes in a chassis are electrically interconnected and form a 1x1x3x2x1x2 grid. Inter-chassis connections are optical and each CMB loads eight optical modules. The Post-FX10 chassis is 2U sized and can be mounted in a standard 19-inch rack. A triplet of chassis is a building block and forms a 1x1x3x2x3x2 grid. A maximum of six building blocks can be placed in a rack; therefore, a Post-FX10 rack mounts a maximum of 216 nodes and doubles the packaging density from that of FX10. A Post-FX10 system scales to hundreds of racks with the high scalability of Tofu2.

On such a large system, reducing communication overhead is critical for the scalability of parallel applications. Tofu2 provides not only enhanced link throughput but also latency reduction and hiding features.

The goal and contributions of this paper are to introduce an architectural overview of Tofu2, discuss the advantages of the SoC integration, and describe the new features of Tofu2. The architectural overview of Tofu2 and enhancements from its predecessor, Tofu1, are presented in Section 2. Section 3 describes the new features of Tofu2 that reduce the overheads of communications and offload collective communications. Section 4 presents the concluding remarks.

2 Architectural Overview

This section presents the architectural overview of Tofu2 and describes the differences between Tofu1 and Tofu2.

2.1 I/O Signal Reduction and Transmission Technology

A FX10 node uses a total of 144 lanes of differential I/O signals for Tofu1: a processor chip has 32 lanes of differential signals to connect an ICC chip, and an ICC chip has 32 lanes for a processor chip and 80 lanes for 10 links. In contrast, a Post-FX10 node uses only 40 lanes for Tofu2. However, for each processor SoC, the number of I/O signals for the interconnect increases 25%.

Although the signal count per link decreases from 8 to 4 lanes, Tofu2 increases the peak link bandwidth from 5 to 12.5 GB/sec. The data transfer rate increases from 6.25 to 25.78125 Gbps, and the encoding scheme is also enhanced from 8b/10b to 64b/66b. Tofu2 uses optical fibers to support 25-Gbps high-speed signal transmission between chassis.

A unit of link-level transfer is called a frame. The minimum frame length is 64 bytes and the maximum is 2,048 bytes. A frame can contain one end-to-end transport packet and two link-level control packets. In Tofu1, each transport or control packet is a unit of link-level transfer. As in Tofu1, received transport packets are transferred via a virtual cut-through switching technique that buffers packets only when the destination link is blocked. The link-level retransmission scheme for error correction is also inherited from Tofu1.

The routing and switching logic of Tofu2 operates at a 390.625-MHz clock frequency, which is an increase of 25% from that of Tofu1. To optimize power consumption, the implementation of Tofu2 doubles the bit width of data-paths from 128 to 256 bits instead of doubling the clock frequency. The frame format of Tofu2 is 32-byte aligned, and is different from the 16-byte alignment of Tofu1.

2.2 Network Topology

The network topology of Tofu2, as well as Tofu1, is 6D mesh/torus. Fig. 2 shows a conceptual model of the 6D mesh/torus network topology. The address system of the

6D mesh/torus network comprises three scalable coordinate axes and the same number of short coordinate axes with fixed size. The scalable axes are X, Y, and Z, and the fixed sized axes are A, B, and C. The sizes of the short A-, B- and C-axes are fixed; 2, 3 and 2, respectively. The short axes interconnect nodes in a full mesh; however, a full mesh between a pair of nodes is just a point-to-point connection, and a full mesh between three nodes is topologically identical to a ring. The scalable axes interconnect nodes in a series or a ring. The lengths of the X- and Y-axes vary depending on the system size, and the length of the Z-axis is defined by a system model. In Post-FX10, the size of the Z-axis is restricted to multiples of three because a building block is a chassis that forms a $1(X) \times 1(Y) \times 3(Z) \times 2(A) \times 3(B) \times 2(C)$ grid.

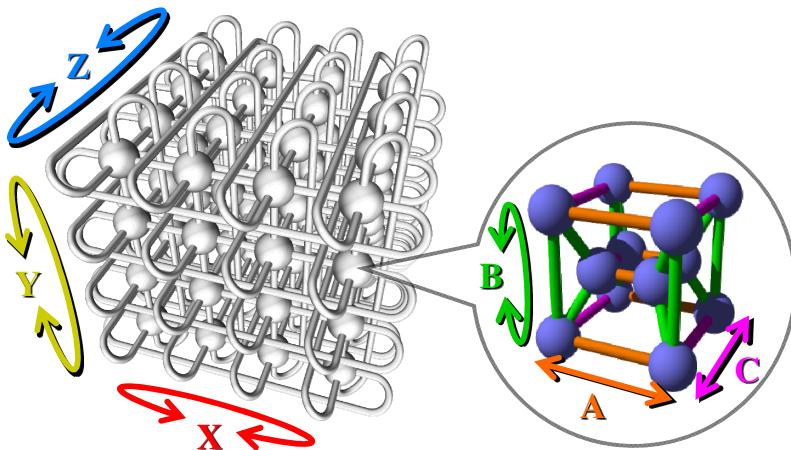


Fig. 2. Conceptual model of 6D mesh/torus network topology

System resource management software divides the 6D mesh/torus network into rectangular submeshes and allocates a submesh for each parallel processing job. Each job is partitioned not to interfere with other jobs' intra-job communications.

Fig. 3 presents simple examples of partitioning on an $8(X) \times 2(A)$ 2D mesh/torus network. The network is partitioned along the X-axis and divided into 3×2 and 5×2 submeshes in example (a) or into four 2×2 submeshes in example (b). Each two-dimensional submesh forms a ring or a ring with chords, although it contains no ring along the X-axis. In addition, there is no restriction in the length along the Z-axis for a submesh despite the restriction on system size.

2.3 Rank Mapping

The rank mapping scheme of Tofu2 uses a combination of a scalable axis and a short axis to form a ring. Three combinations of scalable and short axes provide virtual 3D torus rank mapping. This virtual torus rank mapping scheme allows topology-aware tunings of communication such as multi-dimensional nearest neighbor exchange or further sophisticated optimizations.

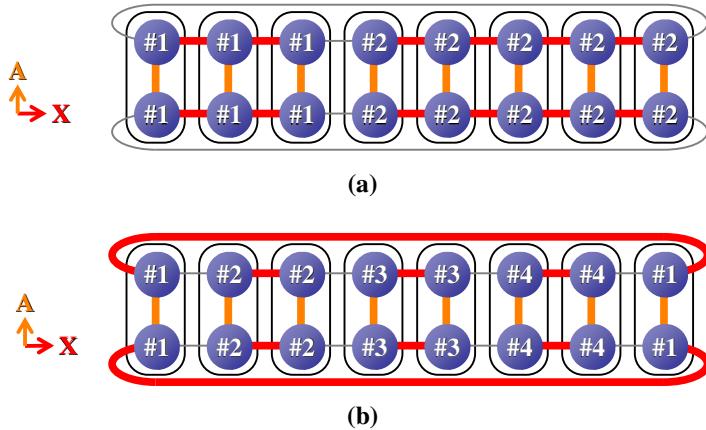


Fig. 3. Examples of partitioning of 8x2 2D mesh/torus network; (a) 3x2 and 5x2 2D meshes (b) four 2x2 2D meshes

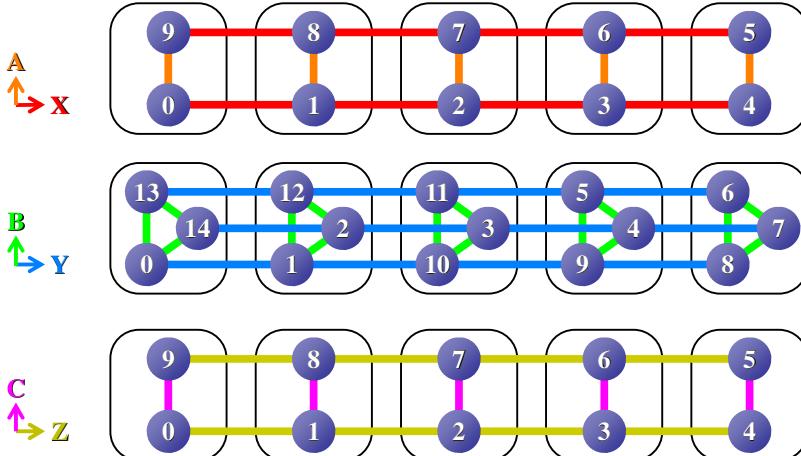


Fig. 4. Example of virtual 3D torus rank mapping; 10x15x10 virtual 3D torus rank is mapped on 5x5x5x2x3x2 submesh

Fig. 4 shows an example of rank mapping on a 5(X)x5(Y)x5(Z)x2(A)x3(B)x2(C) submesh. The rank map forms a 10x15x10 virtual 3D torus. Forming a ring on a 5(Y)x3(B) space requires a bit complicated numbering scheme. It requires a ‘twist’ move along the B-axis at every two hops along the Y-axis.

The virtual torus rank mapping scheme also contributes to system availability and fault-tolerance by enabling utilization of a partition including a failure. It is possible to do nearest neighbor numbering to avoid a single node. If a combination of a scalable axis and a short axis contains the B-axis, it is possible to avoid a single node in nearest neighbor numbering. Fig. 5 shows an example of numbering nodes to avoid a failed node. Example (a) uses all 15 nodes and the other example isolates the bottom node in the middle group from numbering.

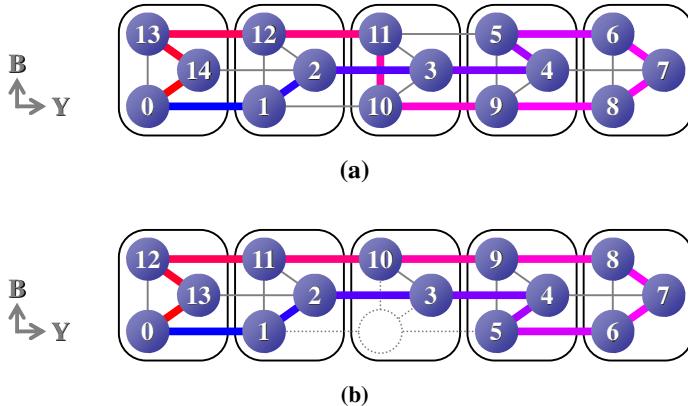


Fig. 5. Examples of virtual torus rank mapping (a) before and (b) after failure isolation

In Post-FX10, a combination of the Z-axis and B-axis is not recommended, because an isolation of a 3(Z)x1(B) area is required because a field replacement unit is a chassis that has a length of three along the Z-axis.

2.4 Communication Engines

The communication engines of Tofu2 are called Tofu network interface (TNI) and Tofu barrier interface (TBI). The TNI enables data transfer to and from another node, and TBI enables inter-node synchronization. A node of Post-FX10, as well as FX10, comprises four TNIs and one TBI.

Four TNIs of a node operate as independent quad-rails and enhance throughput of certain types of traffic patterns, such as nearest-neighbor exchange, pipeline transfer, and multipath transfer. The communication function of the TNI is remote direct memory access (RDMA) which includes Put, Get and newly supported atomic read-modify-write (RMW). Put transfers data to another node, Get transfers data from another node, and atomic RMW carries out an atomic operation on data of another node. Each TNI has eleven sets of control registers. Each register set can be mapped to an arbitrary address space individually to bypass the OS and to eliminate mutual execution among cores. The TNI uses a command queuing mechanism to make communication functions non-blocking. A set of queues including a command queue is called a control queue (CQ), which is laid out on the main memory and manipulated by the control registers.

The TBI has eight sets of control registers and simultaneously synchronizes eight independent groups of nodes. Each set of control registers is called a barrier channel, and can be mapped to an address space of an arbitrary process. The TBI handles barrier synchronization through hardware; therefore, the synchronization delay is not affected by OS jitter which happens to involve a long delay of milliseconds. A barrier channel can also carry out reduction operation and broadcast a result during barrier synchronization. The supported operation types of reduction are floating-point

summation, integer summation, integer maximum value, and bitwise and/or/xor. The floating-point summation of the TBI avoids rounding errors and returns the same result to all nodes. The algorithm of floating-point summation has been introduced in the highly functional switch of Fujitsu's FX1 supercomputer for the first time.

3 New Features

3.1 Mutual and Strong Atomicity for Atomic Read-Modify-Write

In Post-FX10, SPARC64TM XIfx processor and Tofu2 cooperatively introduce mutual and strong atomicity into their atomic operations. The processor's atomic operations and atomic RMW communication functions mutually guarantee atomicity, and the guaranteed atomicity is strong because even memory accesses other than atomic operations cannot break the atomicity. The mutual strong atomicity enables the merging of multi-process and multi-thread synchronization into one shared variable. Mutual and strong atomicity reduces the synchronization overhead between multi-process and multi-thread runtimes and even has the potential to enable an efficient unified multi-process multi-thread runtime or a new parallel language runtime.

A number of conventional HPC interconnects such as InfiniBand [6] or Cray Aries [7] support atomic communication functions. Conventional atomic communication functions guarantee the atomicity among concurrent communication functions, however they do not guarantee atomicity if a processor simultaneously accesses the same data. Therefore, a multi-process synchronization on a shared variable and a multi-thread synchronization on a shared variable must be implemented separately.

3.2 Session-Mode Control Queue

To offload collective communications, Tofu2 introduces a session mode to a CQ of the TNI. A command enqueued into a session-mode CQ does not start immediately but after the session offset of the CQ overtakes the command. The session offset is advanced by session progress step (SPS) on successful reception of the Put transfer, also including 0-byte Put transfer. The session offset can be advanced before commands are enqueued. When there is not enough command in a session-mode CQ for its session offset, a newly enqueued command can start immediately. Therefore each node can independently start a collective communication without blocking. There is no extra overhead when there is not enough command in the receiver-side CQ. Therefore single session-mode CQ is encouraged to be used for multiple types of collective communication unless otherwise the collective communications are overlapped or executed out-of-order. If there is an extra overhead when a receiver is not ready, it potentially requires pre-post of all commands and reuse of the same communicator and the same type of collective communication [8].

Fig. 6 shows an example of offloading a broadcast communication using session-mode CQs. The commands are enqueued in advance and no processor is involved during the broadcast. A message is broadcasted from node 0. The message is divided into four chunks and transferred in a bucket brigade manner. Node 0 transfers four

chunks in succession to fully use the link bandwidth, and the pipelined data transfer reduces the serialization latency at each hop. Node 1 and 2 use a session-mode CQ. Each Put transfers from node 0 carries SPS 1, and each reception of the transfer triggers one command at node 1. Each Put transfer from node 1 also carries SPS 1 and triggers one command at node 2. Each Put transfer from node 2 carries zero SPS.

A chain of Put transfers across session-mode CQs can branch off using 2 or more SPSs, and a chain also can join another chain using the No Operation (NOP) command. Fig. 7 shows an example of offloading a gather communication, which includes branches and joins of Put transfer chains. Node 0 receives data from all other nodes individually in a handshaking manner. A handshaking data transfer requires a

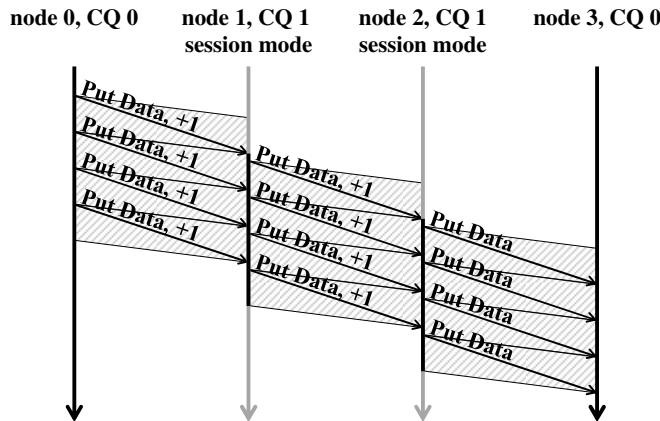


Fig. 6. Example of offloading pipelined Broadcast communication

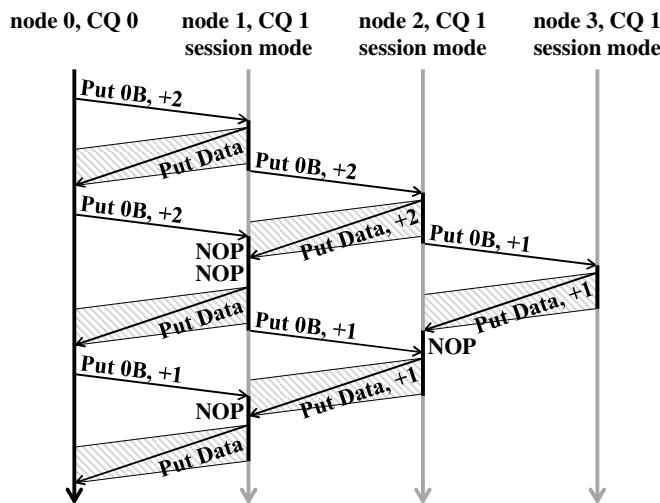


Fig. 7. Example of offloading handshaking Gather communication

small intermediate communication buffer. Nodes 1, 2, and 3 use a session-mode CQ. The first 0-byte Put transfer from node 0 carries SPS 2 and triggers two Put transfers at node 1. The first Put transfer carries data to node 0, and the next Put transfer carries 0 bytes of data and SPS 2. After the transmission of the two Put transfers, the session-mode CQ of node 1 will receive a Put transfer with SPS 2 from node 0 and the same from node 2. No matter which Put transfer is received first, the earlier SPS 2 triggers two NOP commands and the later SPS 2 triggers two Put transfer functions.

3.3 Harmless Cache Injection

The cache coherency has been a brick wall in terms of communication latency because most general purpose processors do not allow I/O devices to write data directly to caches. The SoC design moves processors to accept special tweaks for integrated devices. Tofu2 introduced cache injection and padding techniques to break through the cache wall in cooperation with the processor's cache controller.

The major issues of cache injection are the selection of a cache from private and shared multi-level caches, the control of cache pollution, and the partial write of a cache line. The cache injection technique of Tofu2 addresses these issues with a simple scheme. The received data is injected only to a cache that already has the corresponding cache line, and the cache line is in the exclusive state. Data in an exclusive cache line is highly likely to be polled by a corresponding processor core. An attempt of cache injection is made only on data that are indicated to be injected by the sender and contain a full cache line. A partial update of a cache line is avoided because it requires an additional lock mechanism of cache lines to hold up the coherence protocol during the read-modify-write on the cache line. This simple cache injection scheme helps the processor's cache controller to be extended carefully not to break cache coherency without introducing a complicated mechanism. The additional padding flag is used to optimize the cache injection. If data is small, transferring the single cache line size of data is redundant. The padding flag indicates that the received data should be zero-padded to the cache line boundary at the receiver side.

We estimated reduced communication latency by the cache injection feature. The evaluations used the Verilog RTL codes for the production and measured the communication latencies from the simulated waveforms. The test programs were executed on the simulated processor cores, included no communication software stack like an MPI library, and used Tofu2 hardware directly to minimize the software overhead. Table 1 lists the results of Ping-Pong communication using Put transfers. The estimated half round-trip latencies were 0.87 usec with the injection flag off and 0.71 usec with the injection flag on. The estimated latency with the injection flag off was near the same as the measured latency on an actual FX10 system, which is 0.91 usec [3].

Table 1. Estimated latencies of Put Ping-Pong communication using RTL-level simulation

Injection flag	Half round-trip latency [usec]
Off	0.87
On	0.71

4 Conclusion

We introduced enhanced architecture and specifications of Tofu2 and described new features. Tofu2 inherits the 6D mesh/torus network topology from its predecessor and increases the link throughput by two and half times. Tofu2 is integrated into a newly developed SPARC64TM XIfx processor chip and takes advantages of SoC implementation by removing off-chip I/O between processor and ICC chips. Tofu2 also introduces new features such as the atomic RMW communication functions, session-mode CQ for the offloading of collective communications, and harmless cache injection technique to reduce communication latency.

References

1. Fujitsu Limited: Advanced technologies of the Supercomputer PRIMEHPC FX10 (2012), <http://www.fujitsu.com/downloads/TC/primehpc/primehpc-fx10-hard-en.pdf>
2. Ajima, Y., Sumimoto, S., Shimizu, T.: Tofu: A 6D Mesh/Torus Interconnect for Exascale Computers. *IEEE Computer* 42(11), 36–40 (2009)
3. Ajima, Y., Inoue, T., Hiramoto, S., Shimizu, T., Takagi, Y.: The Tofu Interconnect. *IEEE Micro* 32(1), 21–31 (2012)
4. Ajima, Y., Inoue, T., Hiramoto, S., Shimizu, T.: Tofu: Interconnect for the K computer. *Fujitsu Scientific and Technical Journal* 48(3), 280–285 (2012)
5. Miyazaki, H., Kusano, Y., Shinjou, N., Shoji, F., Yokokawa, M., Watanabe, T.: Overview of the K computer System. *Fujitsu Scientific and Technical Journal* 48(3), 255–265 (2012)
6. InfiniBand Trade Association: InfiniBandTM Architecture Specification Volume 1 Release 1.2.1 (2007)
7. Faanes, G., et al.: Cray cascade: a scalable HPC system based on a Dragonfly network. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC 2012)*, Article 103 (2012)
8. Graham, R.L., et al.: Overlapping computation and communication: Barrier algorithms and ConnectX-2 CORE-Direct capabilities. In: *IEEE International Symposium on Parallel & Distributed Processing, Workshops and Ph.D. Forum (IPDPSW)* (2010)

Compression by Default – Reducing Total Cost of Ownership of Storage Systems

Michael Kuhn, Konstantinos Chasapis, Manuel F. Dolz, and Thomas Ludwig

University of Hamburg, Germany
firstname.lastname@informatik.uni-hamburg.de

1 Introduction and Motivation

High performance I/O is a major stumbling block to reach the ExaFLOPS barrier. While CPU speed and HDD capacity have increased by roughly a factor of 1,000 every 10 years [3], HDD speeds have only seen a 300-fold throughput increase over the last 25 years. To make the problem worse, the growth of HDD capacity has recently also started to slow down, requiring additional investment to keep up with the increasing processing power. Due to the increasing electricity footprints, energy used for storage represents an important portion of the total cost of ownership (TCO) [2]. One way to reduce the amount of required storage hardware and thus the TCO is to employ compression.

2 TCO Model

We aim to provide a model to estimate the benefits of compression in advance. Currently, the model only takes costs for the actual storage hardware into account but not possible indirect savings such as decreased cooling and space requirements. We have already analyzed several compression algorithms in conjunction with climate data and shown that compression ratios of 1.5 and more are possible in this case [1]. One of the most important factors of compression are CPU and power consumption overhead. While modern light-weight algorithms such as `lz4` produce negligible overhead, algorithms such as `gzip` provide higher compression ratios but significantly increase power consumption, making them unsuitable for our use case. We have leveraged Lustre’s ZFS backend; it provides transparent compression and thus requires no modifications to the applications. Additionally, application performance is not influenced because all compression is performed on the servers.

Our TCO model uses a number of different variables to make it possible to adapt it to different storage system requirements: the lifetime of the storage system t (h), the total number of disks n , the number of disks per server n_s , the price per disk c_d (€), the price per server c_s (€), the power consumption per disk p_d (kW), the power consumption per server p_s (kW), the energy price c_e (€/kWh) and the compression ratio r . Using compression algorithms with

negligible overhead (for example, `lz4`) allows using p_d and p_s for both the uncompressed and compressed cases. The TCO can be split up into two parts: Procurement costs (1) and running costs (2).

$$\left\lceil \frac{n}{r} \right\rceil \cdot c_d + \left\lceil \frac{n}{n_s \cdot r} \right\rceil \cdot c_s \quad (1)$$

$$\left(\left\lceil \frac{n}{r} \right\rceil \cdot p_d + \left\lceil \frac{n}{n_s \cdot r} \right\rceil \cdot p_s \right) \cdot t \cdot c_e \quad (2)$$

We use the following values to test our TCO model: $t = 48 \cdot 30 \cdot 24 \text{ h}$ (48 months), $n = 10,000$, $n_s = 64$, $c_d = 200 \text{ €}$, $c_s = 2,000 \text{ €}$, $p_d = 0.01 \text{ kW}$, $p_s = 0.2 \text{ kW}$, $c_e = 0.15 \text{ €/kWh}$ and $r = 1.5$ [1]; the uncompressed case uses $r = 1$. The results for uncompressed and $r = 1.5$ are shown in Figure 1.

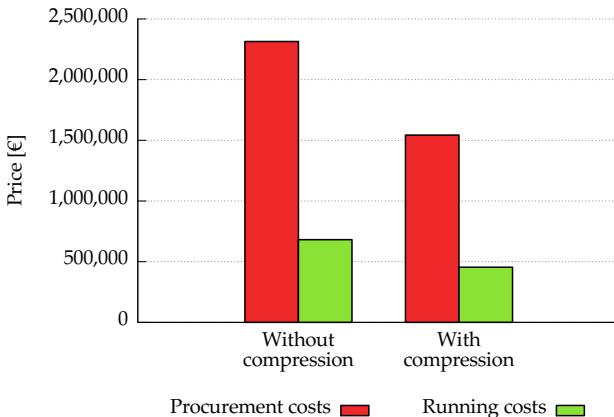


Fig. 1. TCO model results

3 Conclusions and Future Work

Compression in HPC storage servers can be used to reduce costs. We propose to enable compression by default as high compression ratios directly correspond to high savings. However, it remains important to carefully choose compression algorithms due to their inherent CPU overhead. We have identified `lz4` as a suitable compression algorithm for climate data because it produces no significant increase in CPU utilization. We plan to improve the TCO model to take more factors such as different compression algorithms and indirect savings into account. Benefits for different data sets also have to be analyzed.

Acknowledgements. We would like to thank the German Helmholtz Association's LSDMA project and the EU's EXA2GREEN project (FP7-318793) for their support.

References

1. Chasapis, K., Dolz, M.F., Kuhn, M., Ludwig, T.: Evaluating Power-Performance Benefits of Data Compression in HPC Storage Servers. In: The Fourth International Conference on Smart Grids, Green Communications and IT Energy-Aware Technologies (ENERGY 2014) (accepted for publication, 2014)
2. Curry, M.L., Ward, H.L., Grider, G., Gemmill, J., Harris, J., Martinez, D.: Power use of disk subsystems in supercomputers. In: Proceedings of the Sixth Workshop on Parallel Data Storage, pp. 49–54. ACM (2011)
3. Freitas, R., Slemente, J., Sawdon, W., Chiu, L.: GPFS scans 10 billion files in 43 minutes. IBM Advanced Storage Laboratory. IBM Almaden Research Center 95120 (2011)

Predictive Performance Tuning of OpenACC Accelerated Applications

Shahzeb Siddiqui and Saber Feki

King Abdullah University of Science and Technology, Kingdom of Saudi Arabia
{Shahzeb.Siddiqui, Saber.Feki}@kaust.edu.sa

Abstract. GPUs are gradually becoming mainstream in supercomputing as their capabilities to significantly accelerate a large spectrum of scientific applications have been identified and proven. Moreover, with the introduction of directive based programming models such as OpenACC, these devices are becoming more accessible and practical to use by a larger scientific community. However, performance optimization of OpenACC applications usually requires an in-depth knowledge of the hardware and software specifications. We suggest a prediction-based performance tuning mechanism to quickly tune OpenACC parameters to dynamically adapt to the execution environment on a given system. This approach is applied to a finite difference kernel to tune the OpenACC gang and vector clauses for mapping the computations into the underlying accelerator architecture. Our experiments show a good performance improvement against the default compiler parameters and a faster tuning by an order of magnitude compared to the brute force search tuning.

1 Predictive Performance Tuning Methodology

Tuning OpenACC gang and vector clauses [1] with an exhaustive search in the full parameter space is time consuming. We suggest here a tuning methodology based on knowledge of the tuning results of previously tuned problem sizes. The closest problem size in such database is identified and its gang and vector tuned values are used to define a subset of the parameters space to be explored by the tuning engine. The tuning results of the recently investigated problem size are then included to enrich the tuning knowledge database. This approach is proven to work well in tuning MPI communications in the Abstract Data and Communication Library [2].

2 Experimental Results

We present here the performance results of applying the suggested tuning methodology on the RTM isotropic modeling kernel executing on K20c NVIDIA GPUs. As shown in Figure 1, the tuning procedure using either the brute force search or the predictive method resulted in a better performance than the compiler default tuning. A performance increase of up to 80% is recorded against the performance of the compiler-tuned code. Figure 2 shows the required time for tuning a given problem

size with the brute force and the predictive tuning methods. Our experiment shows that the tuning time is reduced dramatically (18X to 52X) by using the predictive tuning approach at the same time a comparable performance to the brute force search approach is achieved.

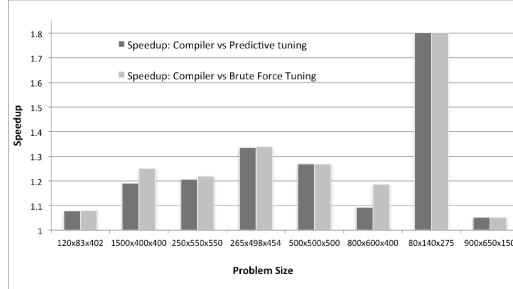


Fig. 1. Performance speedup analysis using the different tuning methodologies against the compiler tuning

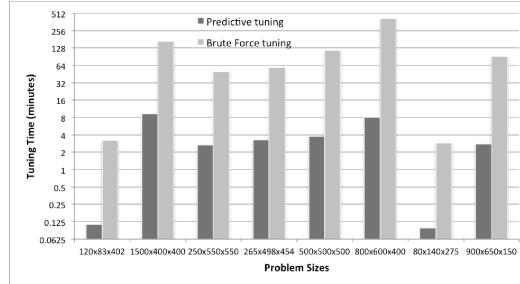


Fig. 2. Tuning time for brute force versus predictive tuning

3 Conclusions and Future Work

A prediction-based approach to perform runtime performance tuning for OpenACC accelerated applications is suggested. The performance results obtained by our tuning methodology on a finite difference kernel showed a significant performance gain against the compiler-tuned code. Furthermore, the time needed for tuning is reduced by an order of magnitude compared to the naive brute force search technique. Our future work includes the automation of the tuning process and its application at runtime to a variety of scientific applications and to the latest accelerator architectures

References

1. OpenACC Standard, <http://www.openacc-standard.org/>
2. Feki, S., Gabriel, E.: A Historic Knowledge Based Approach for Dynamic Optimization. In: Proceedings of the International Conference on Parallel Computing, pp. 389–396 (2009)

Particle-in-Cell Plasma Simulation on CPUs, GPUs and Xeon Phi Coprocessors

Sergey Bastrakov¹, Iosif Meyerov¹, Igor Surmin¹, Evgeny Efimenko²,
Arkady Gonoskov^{1,2}, Alexander Malyshev¹, and Mikhail Shiryaev¹

¹ Lobachevsky State University of Nizhni Novgorod, Russia

² Institute of Applied Physics, Russian Academy of Sciences,
Nizhni Novgorod, Russia

Simulation of plasma dynamics with the Particle-in-Cell method is one of the currently high-demand areas of computational physics. Solving up-to-date physical problems often requires large-scale plasma simulation. Given the growing popularity of GPGPU and the advent of Intel Xeon Phi coprocessors, there is an interest in high-performance implementation of the method for heterogeneous systems.

PICADOR [1] is a tool for three-dimensional fully relativistic plasma simulation using the Particle-in-Cell method. Features of PICADOR include FDTD and NDF field solvers, Boris particle pusher, CIC and TSC particle form factors, Esirkepov current deposition, ionization, and moving frame. The code is capable of running on heterogeneous cluster systems with CPUs, GPUs, and Xeon Phi coprocessors and supports dynamic load balancing. Each MPI process handles a part of simulation area (domain) using a multicore CPU via OpenMP, a GPU via CUDA, or a Xeon Phi coprocessor. All MPI exchanges occur only between processes handling neighboring domains.

The Particle-in-Cell method operates on two major sets of data: an ensemble of charged particles (electrons and ions of various type) and grid values of electromagnetic field and current density. A key aspect of high-performance implementation of the Particle-in-Cell method is obtaining efficient memory access pattern during the most computationally intensive Particle–Grid operations: field interpolation and current deposition. We store particles of each cell in a separate array and process particles in a cell-by-cell order. This scheme helps to improve memory locality and allow vectorization of particle loops. These optimization techniques yield a combined 4x to 7x performance improvement over the baseline implementation.

We employ a widely used performance metric for Particle-in-Cell simulations, that is computational time per particle per time step, namely, nanoseconds per particle update. On a simulation of dense plasma with first-order field interpolation and current deposition in double precision PICADOR achieves 12 nanoseconds per particle update on an 8-core Intel Xeon E5-2690 CPU with 99% strong scaling efficiency on shared memory, which is competitive to the state-of-the-art implementations [2,3].

The Xeon Phi implementation is essentially the same C++/OpenMP code as for CPUs with a minor difference in compiler directives that control vectorization.

A Xeon Phi 7110X coprocessor in native mode scores 8 nanoseconds per particle update on the same benchmark, outperforming the Xeon E5-2690 CPU by factor of 1.5. A heterogeneous Xeon + Xeon Phi combination, with one process running on the processor and another one on the coprocessor, achieves 6 nanoseconds per particle update. However, other heterogeneous configurations, such as 2x Xeon + Xeon Phi or Xeon + 2x Xeon Phi, do not yield any performance benefit due to high MPI exchanges overhead. A major performance drawback on CPUs and particularly on the Xeon Phi is that field component scatter in Yee grid hinders efficient memory access in the vectorized field interpolation.

The GPU implementation employs a variation on the widely used supercell technique [4] with a CUDA block processing particles of a supercell. The main performance challenge in GPU implementation is current deposition, which requires reduction of the results of all threads in a block. We have two implementations of this operation: reduction in shared memory and reduction via atomic operations. The first one appears to be better on Fermi-generation GPUs, while the second is preferable on Kepler-generation GPUs, achieving, respectively, 4x and 10x speedup over 8 CPU cores in single precision.

PICADOR is developed and used by the HPC Center of University of Nizhni Novgorod and the Institute of Applied Physics of Russian Academy of Sciences for simulation of laser-matter interaction. The code architecture is extendable in terms of additional stages and devices and is capable of using modern heterogeneous cluster systems with CPUs, GPUs and Intel Xeon Phi coprocessors. The performance and scaling efficiency are competitive with other implementations. The future prospects include better load balancing between CPUs, GPUs and Xeon Phi, further optimization of GPU and Xeon Phi implementations, development and optimization of additional modules to allow solving a larger set of problems.

The study was supported by the RFBR, research project No. 14-07-31211.

References

1. Bastrakov, S., Donchenko, R., Gonoskov, A., Efimenko, E., Malyshev, A., Meyerov, I., Surmin, I.: Particle-in-cell plasma simulation on heterogeneous cluster systems. *Journal of Computational Science* 3, 474–479 (2013)
2. Fonseca, R.A., Vieira, J., Fiúza, F., Davidson, A., Tsung, F.S., Mori, W.B., Silva, L.O.: Exploiting multi-scale parallelism for large scale numerical modelling of laser wakefield accelerators. *Plasma Physics and Controlled Fusion* 55 (2013)
3. Decyk, V.K., Singh, T.V.: Particle-in-Cell algorithms for emerging computer architectures. *Computer Physics Communications* 185, 708–719 (2014)
4. Burau, H., Widera, R., Honig, W., Juckeland, G., Debus, A., Kluge, T., Schramm, U., Cowan, T.E., Sauerbrey, R., Bussmann, M.: PIConGPU: A Fully Relativistic Particle-in-Cell Code for a GPU Cluster. *IEEE Transactions on Plasma Science* 38, 2831–2839 (2010)

Application Tracking Using the Ichnaea Tools*

Iain Miller, Andy Bennett, and Oliver Perks

AWE Plc., Aldermaston, Reading, RG7 4PR, UK

{iain.miller,andy.bennett,oliver.perks}@awe.co.uk

<http://www.awe.co.uk>

Keywords: Performance evaluation, application tracking, benchmarking.

1 Introduction

High Performance Computing (HPC) platforms are constantly increasing in size and complexity. As they evolve it becomes increasingly difficult to fully utilise their capabilities. In order to improve this utilisation we first need to gather performance data.

This paper introduces the Ichnaea tool suite, an open source set of tools designed to gather performance data from all application runs[1]. The suite also provides the capability to store and analyse these results, including data visualisation.

The Ichnaea tool suite is broken up into several components: a lightweight and flexible application profiling library to gather data through an instrumentation API; a database to store the profiling outputs and a graphical user interface to data mine and visualise the results.

2 Related Work

Most benchmarking environments have elements to simplify the launching of multiple applications and extraction of performance data. Examples of this are the Jülich Benchmarking Environment, JuBE [4], and IBMs JACE [5]. Both systems require users to submit their jobs from within the benchmarking environment, however this is rarely practical for production applications. Another IBM environment is the BlueGene Performance Data Repository [2], that gathers data from every run of an application built with the supplied MPI wrappers. However, it can only collect system and hardware information and is currently only on BlueGene platforms. Profilers, such as Scalasca and Open Speedshop, can also be used to gather the performance data, but these can have a significant impact on runtime and due to the granularity produce a lot of information. Due to these reasons it is impractical to use a profiler on every production run.

* The authors would like to thank John Holt and Jon Hollocombe from Tessella Plc., Simon Hammond from Sandia National Laboratories and Prof. Stephen Jarvis from the University of Warwick for their help in developing PMTM and PMAT. Also, Paul Jelfs, Simon Greene and Martin Nolten at AWE for their help in debugging and feature requests for PMTM.

3 Ichnaea Tool Suite

The first component of the Ichnaea tool suite is the Performance Modelling Timing Module (PMTM) which provides a common interface to system timing routines, such as `gettimeofday()`. Different types of platforms and operating systems may require new internal wrappers to be created, but the external API would remain the same. At the same time it also documents system and code parameters, and environment variables. The outputs are produced in the highly portable and processable Comma Separated Value format. The library is specifically engineered to work in MPI applications and has been shown to scale to hundreds of thousands of cores with minimal performance impact. Recent developments have also made the library threadsafe and suitable for use in hybrid MPI/OpenMP applications. Other functionality allows PMTM to be used in both libraries and host applications and correctly manage the resulting output.

At AWE the hybrid GTr particle tracking application has been updated to utilise PMTM to calculate the cost function for its load balancing algorithm to improve timing accuracy. Changes resulting from a more comprehensive understanding of load imbalance, highlighted by PMTM, have resulted in a performance gain of 10-20%. A PMTM instrumented production code will produce performance data for every user run, with minimal overhead, and allow developers to conduct studies on the effects of various changes.

In order to support developers in these studies the Ichnaea tool suite also includes a relational database structure and the Performance Modelling Analysis Tool (PMAT). PMAT is a Java frontend to the Ichnaea database that can do both data collection and visualisation. PMAT is primarily designed for use with PMTM files but other file formats can be read using custom parsers to add them to the Ichnaea database. PMAT currently includes a parser for the HPCC [3] output files. The GUI allows a user to sift through the data in the database and feed it into GNUPlot for visualisation and can currently produce X-Y, Bar and Kiviat diagrams for viewing and exporting into PNG or GNUPlot formats.

References

1. Ichnaea sourceforge repository, <http://sourceforge.net/projects/ichnaea>
2. Chung, I.H.: Bluegene performance data repository. Presented at ScicomP 2013 (2013), http://spscicomp.org/wordpress/wp-content/uploads/2013/04/chung-BGPDR_scicomp.pdf
3. Dongarra, J.J., Luszczek, P.: Introduction to the hpcchallenge benchmark suite. Tech. rep., DTIC Document (2004)
4. Frings, W., Schnurpeil, A., Meier, S., Janetzko, F., Arnold, L.: A flexible, application-and platform-independent environment for benchmarking. Parallel Computing: From Multicores and GPU's to Petascale 19, 423 (2010)
5. Prabhakar, G., Merchant, S.: An environment for automating hpc application deployment. Presented at ScicomP 2012 (2012), <http://spscicomp.org/wordpress/wp-content/uploads/2012/05/ScicomP-2012-Prabhakar-JACE.pdf>

OpenFFT: An Open-Source Package for 3-D FFTs with Minimal Volume of Communication

Truong Vinh Truong Duy^{1,2} and Taisuke Ozaki¹

¹ Japan Advanced Institute of Science and Technology (JAIST)

1-1 Asahidai, Nomi, Ishikawa 923-1292, Japan

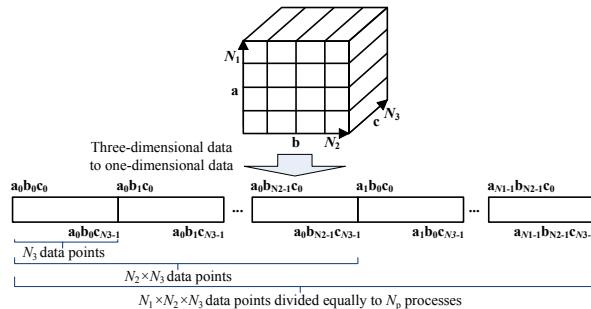
² Institute for Solid State Physics, The University of Tokyo,
Kashiwanoha 5-1-5, Kashiwa, Chiba 277-8581, Japan

Abstract. We develop OpenFFT with a decomposition method for the parallelization of multi-dimensional FFTs possessing two distinguishing features: adaptive decomposition and transpose order awareness for exploiting data reuse when transposing to achieve minimal communication volumes. Based on a row-wise decomposition that translates the multi-dimensional data into one-dimensional data for equally allocating to the processes, our method can adaptively decompose the data in the lowest possible dimensions to reduce communication volume in the first place, differently from previous works that have pre-defined dimensions of decomposition. Also, this decomposition offers plenty of orders in data transpose, and different order results in different volumes of communication. By analyzing all possible cases, we find out the best transpose orders that can reuse data in transpose with minimal communication volumes for 3-D, 4-D, and 5-D FFTs. We implement the method in our OpenFFT package for 3-D FFTs, and numerical results show good performance and scalability of our package.

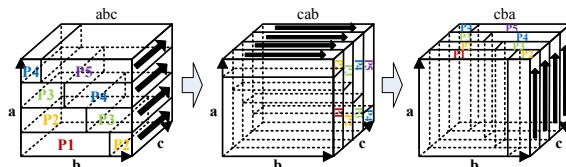
Keywords: 3-D FFTs, multi-dimensional FFTs, domain decomposition, parallel package for FFTs, volume of communication, OpenFFT.

In our method [1], the FFT data is decomposed based on a row-wise basis that maps the M -D data into 1-D data, and translates the corresponding coordinates from multi-dimensions into one-dimension for equally dividing and allocating the resultant 1-D data to the processes. Therefore, the method decomposes in one dimension if the number of processes is less than or equal to the size of one dimension, in two dimensions if the number of processes is greater than the size of one dimension and less than or equal to the size of two dimensions, and so on up to M -dimensions. Figure 1(a) illustrates the decomposition for 3-D FFTs.

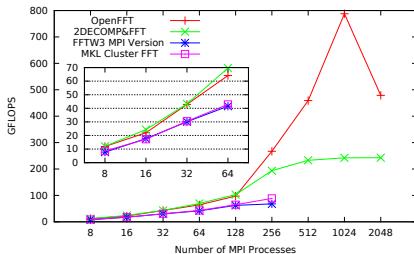
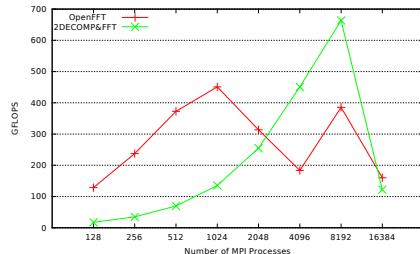
The decomposition results in $(M - 1)!^M$ data transpose orders with different volumes of communication for M -D FFTs. Figure 1(b) exemplifies the operation of our method with transpose-order awareness, where a large amount of data can be reused from *cab* to *cba*. By computationally analyzing all the cases, we find the representative best communication efficient orders: *abc* → *cab* → *cba*, *abcd* → *abdc* → *dcba* → *dcab*, and *abcde* → *abced* → *abedc* → *cedba* → *cedab*, for 3-D, 4-D, and 5-D FFTs, respectively.



(a) 3-D FFTs: 3-D data to 1-D data mapping.



(b) 3-D FFT transpose-order awareness.

(c) Cray XC30 (256³ grid points).(d) K computer (256³ grid points).**Fig. 1.** OpenFFT: Domain decomposition method and numerical results.

We implement the method in our OpenFFT package for 3-D FFTs having C and Fortran interfaces [2], with good performance on various platforms (Figs. 1(c) and 1(d)). The package is employed in our DFT code called OpenMX[3,4].

References

1. Duy, T.V.T., Ozaki, T.: A decomposition method with minimum communication amount for parallelization of multi-dimensional FFTs. *Comput. Phys. Commun.* 185, 153–164 (2014)
2. OpenFFT, <http://www.openmx-square.org/openfft>
3. Duy, T.V.T., Ozaki, T.: A three-dimensional domain decomposition method for large-scale DFT electronic structure calculations. *Comput. Phys. Commun.* 185, 777–789 (2014)
4. OpenMX, <http://www.openmx-square.org>

Author Index

- Aguilera, Alvaro 245
Ajima, Yuichiro 498
Anderson, Matthew 470
Auweter, Axel 394
Ayguadé, Eduard 141, 156
- Bader, Michael 1
Badia, Rosa M. 141
Bastrakov, Sergey 513
Bennett, Andy 515
Bertels, Koen 35
Berzins, Martin 314
Biddiscombe, John 331
Bode, Arndt 394
Bönisch, Thomas 245
Boulton, Michael 53
Brehm, Matthias 394
Bremer, Peer-Timo 314
Breuer, Alexander 1
Brochard, Luigi 394
Brodowicz, Maciej 470
Brown, Nick 460
Browne, James 261
Brugger, Eric 314
Budiardja, Reuben 430
- Carns, Philip 314
Chakraborty, Sourav 278
Chasapis, Konstantinos 508
Checconi, Fabio 109
Chen, Jacqueline 314
Christensen, Cameron 314
Chut, Andriy 245
Ciobanu, Cătălin 487
Crosby, Lonnie 430
Curioni, Alessandro 331
Curran, Dan 53
- Dalessandro, Luke 470
Davies, Craig 487
Davis, Kei 348
DeBuhr, Jackson 470
Delalondre, Fabien 331, 440
de Supinski, Bronis R. 172
- De Zeeuw, Chris I. 487
di Mare, Luca 410
Dolz, Manuel F. 508
Druzhinin, Egor 420
Dubey, Pradeep 124
Duy, Truong Vinh Truong 517
- Efimenko, Evgeny 513
Egawa, Ryusuke 450
Elangovan, Vinoth Krishnan 141
Ewart, Timothée 440
- Fahay, Mark R. 430
Feki, Saber 511
Feng, Wu-chun 172
Fenkes, Joachim 331
Fialho, Leonardo 261
Fiorin, Leandro 35
Fitch, Blake G. 331
Franceschini, Michele M. 331
Fujisawa, Katsuki 365
- Gabriel, Alice-Agnes 1
Germain, Robert S. 331
Gila, Miguel 331
Gonoskov, Arkady 513
Grigori, Laura 76
Grout, Ray 314
- Hadade, Ioan 410
Hagiwara, Takashi 199
Hagleitner, Christoph 35
Hamidouche, Khaled 278
Hammer, Nicolay 394
Hammond, Simon D. 19
Heinecke, Alexander 1
Hiramoto, Shinya 498
Hübbe, Nathanael 245
Huber, Herbert 394
- Ikeda, Yoshiro 498
Inoue, Tomohiro 498
Isobe, Yoko 199

- Jacquelin, Mathias 76
 Januszewski, Radosław 382
 Jenkins, John 296
 Jiang, Song 348
- Kawashima, Takahiro 498
 Khabou, Amal 76
 Kimpe, Dries 296
 Kobayashi, Hiroaki 450
 Kolla, Hemanth 314
 Komatsu, Kazuhiko 450
 Kucuk, Gurhan 187
 Kuhn, Michael 508
 Kumar, Sidharth 314
 Kumbhar, Pramod S. 331
 Kunkel, Julian M. 245
- Labarta, Jesus 156
 Ludwig, Thomas 508
 Lüttgau, Jakob 245
- Malyshev, Alexander 513
 McIntosh-Smith, Simon 53
 McNally, Stephen 430
 Mencer, Oskar 487
 Metzler, Bernard 331
 Meyer, Norbert 382
 Meyerov, Iosif 513
 Michel, Roman 245
 Mickler, Holger 245
 Miller, Iain 515
 Mironov, Vladimir 420
 Miura, Kenichi 498
 Momose, Shintaro 199
 Moriyama, Osamu 498
 Morjan, Peter 331
 Moskovsky, Alexander 420
- Nowicka, Joanna 382
- Okamoto, Takayuki 498
 Ou, Jianqiang 348
 Ozaki, Taisuke 517
- Panda, Dhabaleswar K. 278
 Panda, Raj 394
 Park, Jongsoo 124
 Pascucci, Valerio 314
 Pelties, Christian 1
- Perks, Oliver 515
 Petrini, Fabrizio 93, 109
 Price, James 53
- Que, Xinyu 109
- Rettenberger, Sebastian 1
 Ross, Robert 296
 Rountree, Barry 172
- Samatova, Nagiza F. 296
 Sato, Yukinori 365
 Schmidt, John A. 314
 Schneidenbach, Lars 331
 Schürmann, Felix 331, 440
 Scogland, Thomas R.W. 172
 Scorzelli, Giorgio 314
 Seki, Ken 498
 Semin, Andrey 420
 Shida, Naoyuki 498
 Shimizu, Toshiyuki 498
 Shiryaev, Mikhail 513
 Shmelev, Alexey 420
 Siddiqui, Shahzeb 511
 Smaragdos, Georgios 487
 Smelyanskiy, Mikhail 124
 Solnushkin, Konstantin S. 232
 Sourdis, Ioannis 487
 Sterling, Thomas 470
 Strydis, Christos 487
 Subotic, Vladimir 156
 Subramoni, Hari 278
 Sumimoto, Shinji 498
 Sundaram, Narayanan 124
 Surmin, Igor 513
 Szymanski, Ted H. 215
- Tabata, Takekazu 498
 Tacchella, Davide 331
 Takahara, Hiroshi 199
 Takizawa, Hiroyuki 450
 Tang, Houjun 296
 Thomas, Francois 394
 Thompson, Aidan P. 19
 Trott, Christian R. 19
- Uno, Shunji 498
 Uslu, Gamze 187
- Valero, Mateo 156
 Venkatesh, Akshey 278

- Vermij, Erik 35
Vishwanath, Venkatram 314
Wang, Xuan 245
Ward, T.J. Christopher 331
Weging, Johann 245
Wilde, Torsten 394
Xu, Lei 478
Xu, Ying 478
Yasui, Yuichiro 365
Yesil, Cagri 187
Yoon, Doe Hyun 93
Yoshikawa, Takahide 498
Zhang, Xuechen 348
Zimmer, Michaela 245
Zou, Xiaocheng 296