

High Performance Computing Symposium

(HPC 2015)

2015 Spring Simulation Multi-Conference (SpringSim'15)

Simulation Series Volume 47 Number 4

**Alexandria, Virginia, USA
12 – 15 April 2015**

Editors:

**Layne T. Watson
Joseph Weinbub
Masha Sosonkina**

**William I. Thacker
Karl Rupp**

ISBN: 978-1-5108-0101-1

Printed from e-media with permission by:

Curran Associates, Inc.
57 Morehouse Lane
Red Hook, NY 12571
www.proceedings.com



Some format issues inherent in the e-media version may also appear in this print version.

© 2015 SIMULATION COUNCILS, INC.

Responsibility for the accuracy of all statement in each paper rests solely with the author(s). Statements are not necessarily representative of, nor endorsed by, The Society for Modeling and Simulation International.

Printed by Curran Associates, Inc. (2015)

Permission is granted to photocopy portions of this publication for personal use and for the use of students provided credit is given to the conference and publication. Permission does not extend to other types of reproduction nor to copying for incorporation into commercial advertising nor for any other profit-making purpose. Other publications are encouraged to include 300- to 500-word abstracts or excerpts from any paper contained in this book, provided credits are given to the author and the conference. For permission to publish a complete paper write: The Society for Modeling and Simulation International (SCS), 2598 Fortune Way, Suite I, Vista, CA 92081, USA.

Additional copies of the Proceedings are available from:

Curran Associates, Inc.
57 Morehouse Lane
Red Hook, NY 12571
curran@proceedings.com
www.proceedings.com/0128.html

or

The Society for Modeling
and Simulation International
2598 Fortune Way, Ste I
Vista, CA 92081 USA
www.scs.org

ISBN: 978-1-5108-0101-1
PRINTED IN THE UNITED STATES

TABLE OF CONTENTS

Towards a More Fault Resilient Multigrid Solver	1
<i>J. Calhoun, L. Olson, M. Snir, W. Gropp</i>	
Exploiting Computing Power of Xeon and Intel Xeon Phi for a Molecular Dynamics Application	9
<i>B. Mathew, N. Rai, A. Gupta, A. Harode</i>	
Fast Parallel Conversion of Edge List to Adjacency List for Large-Scale Graphs	17
<i>S. Arifuzzaman, M. Khan</i>	
A Research Framework for Exascale Simulations of Distributed Virtual World Environments on High Performance Computing (HPC) Clusters	25
<i>A. Goel, W. Karwowski, W. Rivera, M. Montgomery, P. Kincaid, N. Finkelstein</i>	
Fast Sparse Matrix Multiplication on GPU	33
<i>L. Polok, V. Ila, P. Smrz</i>	
ExaShark: A Scalable Hybrid Array Kit for Exascale Simulation	41
<i>I. Chakroun, T. Haber, T. Vander, R. Wuyts, B. Fraine, W. Demeuter</i>	
A Load Balancing Parallel Method for Frequent Pattern Mining on Multi-core Cluster	49
<i>L. Vu, G. Alaghband</i>	
Efficient Scaling of a Hydrodynamics Simulation Using Compiler-based Accelerator Technology	59
<i>J. Bradshaw, P. Moore, B. Torkian</i>	
Sharer Status-based Caching in Tiled Multiprocessor Systems-on-Chip	67
<i>P. Damodaran, A. Zaib, S. Wallentowitz, T. Wild, A. Herkersdorf</i>	
Accelerating the LOBPCG method on GPUs using a Blocked Sparse Matrix Vector Product	75
<i>H. Anzt, S. Tomov, J. Dongarra</i>	
Incremental, Distributed Single-Linkage Hierarchical Clustering Algorithm Using MapReduce	83
<i>C. Jin, Z. Chen, W. Hendrix, A. Agrawal, A. Choudhary</i>	
Throughput Studies on an InfiniBand Interconnect via All-to-All Communications	93
<i>N. Mistry, J. Yanchuck, J. Ramsey, X. Huang, B. Wiley, M. Gobbert</i>	
Parallel Performance of Higher-Order Methods on GPU Hardware	100
<i>T. Spilhaus, J. Buckley, G. Khanna</i>	
DOEE: Dynamic Optimization Framework for Better Energy Efficiency	107
<i>J. Haj-Yihia, A. Yasin, Y. Ben-Asher</i>	
Predicting Energy Consumption Relevant Indicators of Strong Scaling HPC Applications for Different Compute Resource Configurations	115
<i>H. Shoukourian, T. Wilde, A. Auweter, A. Bode, D. Tafani</i>	
A Virtual Machine Model for Accelerating Relational Database Joins Using a General Purpose GPU	127
<i>K. Angstadt, E. Harcourt</i>	
Performance Analysis and Design of a Hessenberg Reduction using Stabilized Blocked Elementary Transformations for New Architectures	135
<i>K. Kabir, A. Haidar, J. Dongarra, S. Tomov</i>	
Efficient Algorithms for Improving the Performance of Read Operations in Distributed File System	143
<i>T. Krishna, T. Ragunathan, S. Battula</i>	
Long-time Simulation of Calcium Induced Calcium Release in a Heart Cell using the Finite Element Method on a Hybrid CPU/GPU Node	150
<i>X. Huang, M. Gobbert</i>	
High Performance Kirchhoff Pre-Stack Depth Migration on Hadoop	158
<i>C. Li, Y. Wang, H. Yan, C. Zhao, J. Zhang</i>	
Parallel QR Algorithm For The C-Method: Application To The Diffraction By Gratings And Rough Surfaces	166
<i>C. Pan, N. Emad, R. Dusseaux</i>	
A Study Of Manycore Shared Memory Architecture As A Way To Build SOC Applications	174
<i>Y. Asher, Y. Shajrawi, Y. Gendel, G. Haber, O. Segal</i>	
Solving The Klein-Gordon Equation Using Fourier Spectral Methods: A Benchmark Test For Computer Performance	182
<i>S. Aseeri, B. Leu, B. Muite, M. Quell, R. Speck, O. Batrasev, A. Liu, E. Muller, H. Servat, M. Moer, M. Icardi, N. Li, B. Palen, P. Sheth, J. Vienne</i>	
A Self-Adaptive Method for Frequent Pattern Mining using a CPU-GPU Hybrid Model	192
<i>L. Vu, G. Alaghband</i>	
Computational Steering for High Performance Computing Applications on Blue Gene/Q System	202
<i>B. Danani, B. D'Amora</i>	

Strategies to Hide Communication for a Classical Molecular Dynamics Proxy Application	210
<i>I. Ngatang, M. Sosonkina</i>	
Shared-Memory Parallelization of the Semi-Ordered Fast Iterative Method	217
<i>J. Weinbub, F. Dang, S. Selberherr, T. Gillberg</i>	
PerDome: A Performance Model for Heterogeneous Computing Systems	225
<i>L. Tang, X. Hu, R. Barrett</i>	
An Improved Probability-One Homotopy Map for Tracking Constrained Clustering Solutions	233
<i>D. Easterling, L. Watson, N. Ramakrishnan</i>	
Productive Parallel Programming with CHARM++	241
<i>P. Miller</i>	
Author Index	

Welcome from the General Chair

Dear Colleagues and friends,

On behalf of the Organizing Committee, I am delighted and honored to welcome you to the Spring Simulation Multi-Conference 2015 (SpringSim'15) in Alexandria, Virginia.

The Society for Modeling & Simulation International (SCS) which organizes Springsim is one of the oldest Modeling and Simulation organizations in the world. It endeavors to promote the advancement of Modeling and Simulation and connect Modeling and Simulation professionals worldwide. **The SpringSim'15 program** includes a world-class selection of peer-reviewed paper, presentations, distinguished keynote speeches and tutorials. In addition, SpingSim'15 offers 1) a poster track for students to showcase their work and receive feedback 2) a Ph.D. colloquium where students and established professionals can meet and exchange ideas and 3) for the first time a student **mobile application competition** where they can show how Modeling and Simulation can be meshed with new technologies. Great thanks to the organizations that have donated money, licenses and books to recognize the best submissions at this conference.

I would also like to thank our keynote speakers **Prof. Wesley Wildman**, **Prof. Sally Brailsford** and **Mr. Jesse Citizen** for graciously accepting to share their vast knowledge and experiences with us. My thanks also go to all members of the Organization Committee for their tireless efforts especially in working through the introduction of a new format and editing process. It was truly a team effort. The committee consists of:

Vice-General Chair	Navonil Mustafee, Exeter University
Program Chair	Saurabh Mittal, Dunip Technologies, LLC
Proceedings Chair	<u>Shafagh</u> Jafer, Embry-Riddle Aeronautical University
Awards Chair	Umut Durak, DLR
Tutorial Chair	Jose Padilla, VMASC
Publicity Chair	Gregory Zacharewicz, Université Bordeaux I
Sponsorship Chair	Andreas Tolk, Simis, Inc.

As a Multi-Conference, our success depends heavily on the track organizers, reviewers and committee members. I am very grateful for the efforts of all of the volunteers that dedicated their time and effort to review and edit all of the submissions and thus make this gathering possible. I also express my gratitude to authors and tutorial presenters for their important contributions.

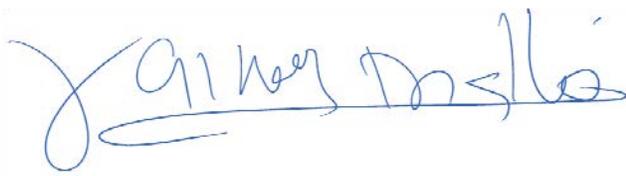
My sincere appreciation goes to the symposia chairs, whose invaluable efforts in their respective sections were key to the success of the overall multi-Conference. This year's symposia chairs are:

- Agent-Directed Simulation (ADS) Symposium, chaired by Levent Yilmaz and Tuncer Ören,
- Communications and Networking Symposium (CNS), chaired by Abolreza Abhari and Hala ElAarag
- High Performance Computing Symposium (HPC), chaired by Layne Watson and Josef Weinbub
- Symposium on Simulation for Architecture and Urban Design (SimAUD), chaired by Shajay Bhooshan and Holly Samuelson
- Theory of Modeling and Simulation (TMS/DEVS), chaired by Fernando Barros and Moon Ho Hwang
- Annual Simulation Symposium (ANSS), chaired by Andreas Tolk and Shafagh Jafer
- Modeling and Simulation in Medicine (MSM) chaired by Jerzy Rozenblit and Johannes Sametinger
- Tutorial and Vendor Track, chaired by Jose Padilla
- Posters session & Student Colloquium, chaired by Salim Chemlal and Mohamed Moallemi
- Work in Progress session chaired by Gregory Zacharewicz

Special thanks go to the SCS officers, **Oletha Darensburg**, **Aleah Hockridge** and the team for their high level of professionalism, and for the smooth running of all the events.

Alexandria is one of the oldest cities in the United States and has an historic *Old Town* with great dining and shopping. In addition, please feel free to sign up for our organized night tour to see the various monuments and cultural centers in Washington, D.C.

Welcome to Springsim'15



Saikou Diallo, Ph.D.

General Chair SpringSim 2015

Old Dominion University, Virginia, Modeling Analysis and Simulation Center

HPC' 15 CHAIRS' MESSAGE

Welcome to the 2015 High Performance Computing Symposium! This is the 23rd special symposium devoted to the impact of high performance computing and communications on computer simulations. The symposium, part of the 2015 Spring Simulation Multi-conference, encompasses a wide variety of topics with a focus on tools and applications for the simulation of physical and engineering systems.

Advances in networking, high end computers, large data stores, special purpose hardware, parallel programming languages, and middleware capabilities are ushering in a new era of high performance parallel and distributed simulations. Along with these new capabilities come new challenges in computing and system modeling. The goal of HPC 2015 is to encourage innovation in high performance computing and communication technologies and to promote synergistic advances in modeling methodologies and simulation. It will promote the exchange of ideas and information between universities, industry, and national laboratories about new developments in system modeling, high performance computing and communication, and scientific computing and simulation. Topics of interest include large scale simulations, numerical methods, problem solving environments, visualization and data management, parallel algorithms for emerging architectures, software tools, and power-aware computing.

The 23rd High Performance Computing Symposium features 27 high quality full papers covering a very wide range of topics, as is evident from a glance at the index of papers, and a tutorial on the parallel programming system Charm++. HPC 2015 would have been impossible without the contributions from many individuals---the symposium organizers who planned and organized everything, the Steering Committee who provided valuable advice and support, and the Program Committee who rigorously reviewed the submissions and their revisions. We have been involved with this symposium since 2000, and this is the best collection of papers we have ever had, thanks to the hard work of the chairs and all the committee members, and the participation of the HPC community. The conference organizers, Steering Committee, and Program Committee hope that everyone enjoys this year's outstanding program, and the Alexandria venue.

Layne T. Watson, General Chair
Josef Weinbub, Vice-general Chair
Masha Sosonkina, Program Chair
William I. Thacker, Vice-program Chair
Karl Rupp, Publicity Chair

SYMPOSIUM ORGANIZERS

General Chair

Layne T. Watson, Virginia Polytechnic Institute & State University

General Vice-Chair

Joseph Weinbub, Vienna University of Technology

Program Chair

Masha Sosonkina, Ames Laboratory & Old Dominion University

Program Vice-Chair

Will Thacker, Winthrop University

Steering Committee

Marc Baboulin, Inria Saclay---Ile-de-France and Universite Paris-Sud

Gary Howell, North Carolina State University

Fang 'Cherry' Liu, Georgia Institute of Technology

Niraj Srivastava, Raytheon Corporation

Program Committee

Alex Aravind, University of Northern British Columbia

Eric Aubanel, University of New Brunswick, Canada

Sanjutka Bhowmick, University of Nebraska

Brett Bode, Ames Laboratory

Ali Butt, Virginia Polytechnic Institute

Bruno Carpentieri, CERFACS, France

Haiyan Cheng, Willamette University

Jing-Ru C. "Ruth" Cheng, U.S. Army Research and Development Center

Jose C. Cunha, Universidade Nova de Lisboa

Matt Dixon, University of San Francisco

Nahid Emad, Universite' de Versailles Saint-Quentin-en-Yvelines, France

Samantha Foley, Oak Ridge National Laboratory
Apala Guha, University of Chicago
Azzam Haidar, ICL, University of Tennessee
Phil Hammonds, RTSync Corporation
Gary Howell, North Carolina State University
Qi Hu, Facebook
Joshua Hursey, Oak Ridge National Laboratory
Jim Jones, Florida Tech
Piotr Luszczek, University of Tennessee, Knoxville
Asif Mahmood, University of Bridgeport
Michael Mascagni, Florida State University
Gabriel Mateescu, Leibniz-Rechenzentrum, Germany
Phil Moore, University of South Carolina
Thomas Rauber, University of Bayreuth, Germany
Jill Reese, Appalachian State University
Cal Ribbens, Virginia Polytechnic Institute
Gudula Ruenger, Technical University of Chemnitz, Germany
Fan Rui, Nanyang Technological University, Singapore
Yang Song, IBM Almaden Research Center
Steve Stevenson, Clemson University
Alan Stewart, Queen's University, Belfast, UK
Peter Tang, Intel Corporation
William A. Ward, CSC, NASA Greenbelt
Qin Xin, Universite' Catholique de Louvain, Belgium
Ping Yang, Pacific Northwest National Lab
Dongping Zhang, AMD Corporation
Yaxiong Zhao, Google

Towards a More Fault Resilient Multigrid Solver

Jon Calhoun
jccalho2@illinois.edu

Luke Olson
lukeo@illinois.edu

Marc Snir
snir@illinois.edu

William D. Gropp
wgropp@illinois.edu

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

ABSTRACT

The effectiveness of sparse, linear solvers is typically studied in terms of their convergence properties and computational complexity, while their ability to handle transient hardware errors, such as bit-flips that lead to silent data corruption (SDC), has received less attention. As supercomputers continue to add more cores to increase performance, they are also becoming more susceptible to SDC. Consequently, understanding the impact of SDC on algorithms and common applications is an important component of solver analysis. In this paper, we investigate algebraic multigrid (AMG) in an environment exposed to corruptions through bit-flips. We propose an algorithmic based detection and recovery scheme that maintains the numerical properties of AMG, while maintaining high convergence rates in this environment. We also introduce a performance model and numerical results in support of the methodology.

Author Keywords

Algebraic Multigrid, Silent Data Corruption, Fault Tolerance, Resilience

INTRODUCTION

Many scientific applications, from modeling blood flow to electromagnetics, depend on sparse matrix structures and linear algebra computations. These types of computations often consume a sizable percent of a high performance computing (HPC) workload. In particular, one crucial operation is the sparse, linear solve. Scalability and convergence of such methods are well studied, but their behavior in the presence of hardware faults is less developed. Current and emerging HPC architectures are expected to experience higher levels of faults than previous architectures and understanding the impact of fault(s) on the *algorithm* is an important component in fully utilizing their resources. Consequently, resiliency techniques need to be developed and analyzed to allow linear solvers the ability to remain efficient and scalable on emerging HPC architectures.

Modern scientific computing relies on solving large, sparse systems of linear equations. Sophisticated solvers are employed to take advantage of this sparsity, and as computing capabilities continue to progress so do the demands on the linear solver. One solver that has shown to be flexible across a range of different architectures is algebraic multigrid (AMG), due to its potential scalability, robustness, and efficiency as an $\mathcal{O}(n)$ complexity method.

As machines are built using a higher number of cores the individual cores themselves are not becoming more reliable [3]. Therefore, as the number of cores in a system increases, the mean time between interruptions decreases. As a result, fault tolerance and resilience are receiving increased attention. Most of this attention is devoted to developing traditional checkpoint-restart libraries [13, 16]. Checkpoint-restart is designed to correct fail-stop errors that do not allow the application to proceed once the error manifests. A main disadvantage of a typical checkpoint-restart scheme is the increase in time and energy to complete a run. Advanced approaches attempt to address these issues, but techniques at the *algorithm* level offer an opportunity to further enhance resilience. Some approaches use mathematical theory and numerical methods in-order to re-construct the missing data due to a fail-stop error [7, 1].

The increase in the number of circuits, coupled with the decrease in feature size, and the use of low power techniques are also likely to increase the frequency of silent data corruption (SDC) and poses a major problem to the future of large scale scientific computing [5, 20]. General methods for SDC detection leverage redundancy [12], but algorithm based fault tolerance (ABFT) has the potential to reduce the overhead for handling SDC by leveraging algorithm dependent heuristics or invariants to detect and recover from SDC [8, 10]. Furthermore, the use of ABFT increases the resiliency of the application while at the same time lowering the time to solution in faulty environments.

The focus of this paper is on utilizing ABFT to improve resiliency of AMG. Understanding the level of resiliency that is provided by AMG on emerging architectures is important for AMG and other sparse, linear solvers.

To motivate the need for silent data corruption (SDC) detection and recovery in the sparse, linear solve, in Figure 1 we consider the residual history for the solution of a diffusion problem in 2D (cf. Section 5). In this example, a single fault is injected into the residual calculation before restriction during the second iteration on the finest level. This fault is a single bit-flip in the first element of the residual vector, a IEEE 754 double precision floating point number, on process 0 of a 16 process job. The fault does not lead to a segmentation fault or *segfault*, and is thus potentially silent, but influential in its impact on the algorithm. Depending on *which* bit is flipped, the number of extra iterations required to achieve the convergence tolerance of $1e^{-7}$ ranges from 0 (bits in the mantissa) to double the original amount (bits in the exponent), thus motivating the need for SDC detection and recovery.

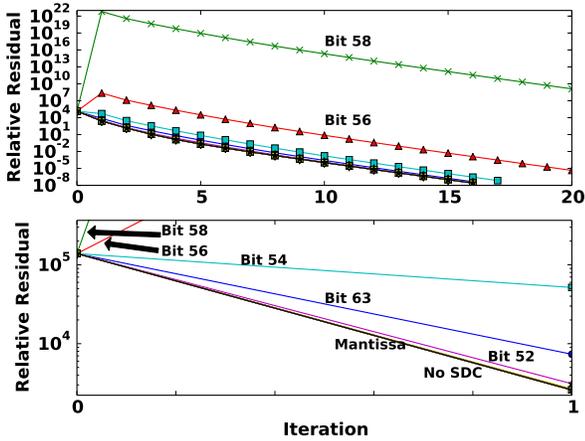


Figure 1. A single bit-flip in a IEEE 754 double precision floating point element in the residual vector’s effect on the iterations needed to converge. In this location, a flipped bit in the exponent causes an increase in the iterations till convergence; while a flipped bit in the mantissa or the sign bit, bit 63, has no effect.

To this end, we improve and analyze the resilience of AMG in the presence of silent faults. In particular, in this paper we make the following contributions:

- low overhead algorithmic based recovery for AMG;
- low cost SDC detectors for iterative linear solvers; and
- AMG specific SDC detectors.

BACKGROUND

Algebraic Multigrid

Consider the sparse matrix problem $Ax = b$. In a parallel setting, iterative solution techniques are preferred due to the memory and complexity requirements to solve. One such approach is algebraic multigrid (AMG) [18], which is often used as a preconditioner for a Krylov method such as conjugate gradient (CG) or generalized minimum residual (GMRES).

AMG constructs a hierarchy of successively coarser problems that are used to iteratively refine the error in the solution. Figure 2 outlines a *solve* cycle of AMG, where an initial guess x^0 is refined using two compatible operations: relaxation, such as weighted Jacobi, and coarse-grid correction, which is a subspace projection method. Relaxation is responsible for reducing high energy error, while coarse grid correction targets *algebraically* smooth error, or error that is invariant to relaxation. The cycle proceeds by successively coarsening the error equations, $Ae = r$, until a coarse level problem is effectively solved in a few steps of relaxation or is efficiently processed with a direct method. Errors on coarse levels are then interpolated to correct solutions. Figure 2 represents the well-known V-cycle and is a common approach to traverse the AMG hierarchy in parallel.

Resilience

Resilience has been a challenge since the early days of computing. Vacuum tube based machines had a mean-time-between-failure (MTBF) of a few days. With the introduction

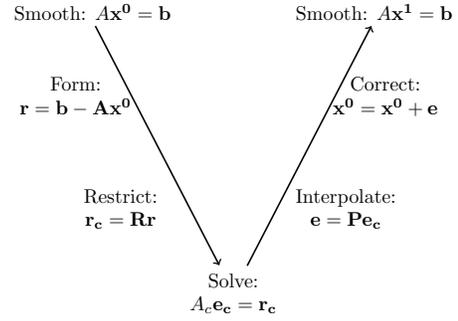


Figure 2. AMG V-cycle process used to solve $Ax = b$

of transistors and integrated circuits, the reliability of the underlying hardware increased, but with the small feature sizes and low-power environments in emerging architectures, fault awareness is making a resurgence [3].

All hardware components do not exhibit the same level of reliability. DRAM and SRAM are susceptible to bit-flips, which has led to error correction codes (ECC) and so-called *chipkill* that protect against bit-flips in memory [21]. These methods, at the cost of area and power on chip, seek to recover the data by stored parity information. In the case of chipkill, a multi-bit errors and in some cases the entire memory chip can be detected and repaired.

The traditional way to handle fail-stop errors is checkpoint-restart [5]. In this approach, the application executes for a given time and saves its state (in full or in part) to permanent storage. In the event of a detected failure, a checkpoint file is read, data structures are rebuilt, and computation is restarted. A variety of checkpoint-restart schemes have been designed, with varying levels of intervention required by the programmer, checkpoint sizes, and checkpoint frequencies considered [16, 13]. In addition, for certain fault considerations and applications, checkpointing is not required [7, 1].

Emerging architectures are demanding attention to silent data corruption (SDC). Extensive work has been done for SDC detection in numerical computations [14, 2, 11, 8], whereas general methods for detection and correction for SDC have also been proposed [19, 9, 12], but have yet to gain wide adoption.

AMG Resilience

The resiliency of AMG to SDC has been studied previously [15], where checksums are used to detect SDCs. The approach is limited to dense matrices, but is robust, incorporating checks for matrix-vector multiplications, relaxation, and interpolation.

More recent work [6] reports that AMG is resilient to SDC due to its iterative and multilevel nature. Provided that AMG does not segfault, a SDC with high probability emerges as an error in the solution vector. This error is subsequently removed at the cost of more work. Moreover, the error is reduced on a coarser level, where this error is more pronounced. The recovery scheme addresses a segfault by triplicating key pointers and ranks the three instances when accessed, accept-

ing the pointer with the most credibility. This is done for every access to the protected arrays, therefore yielding resiliency at a large overhead. Regardless, the approach decreases the number of segfaults for AMG in a faulty environment, yet the resiliency overhead is present even in the case when AMG is not exposed to faults.

In contrast, our recovery scheme induces less overhead, even when AMG does not suffer a segfault. In addition, our scheme provides checks that alert the AMG solver of the presence of SDC and attempts to avoid the impact of the fault instead of allowing AMG to remove the error through more iterations.

TRANSIENT ERROR DETECTION

We first discuss our error model and assumptions. For this method, we assume that memories are sufficiently protected with ECC and chipkill; as such, we do not model SDC arising in memory. Further, we assume that errors emerge during instruction execution in the solve phase of AMG and that corruptions manifest themselves as bit perturbations is the result of instructions. In addition, during the solve phase the operators A , P , and R on each level as well as the right hand side \mathbf{b} are never written only read. As a result, these structures are not checkpointed for SDC recovery since they are only modified through errant stores. To protect these data structures from such writes, one approach is to use the system call `mprotect` to force the associated memory pages to be read-only. In the case of a write to a protected page, a segfault is raised triggering recovery. This function is used in some asynchronous checkpoint-restart approaches [17] to exploit copy on write capabilities when scheduling pages to be checkpointed.

MultiLevel Recovery Scheme

As faults occur inside AMG, we classify them as the following. A fault,

- decreases the convergence time;
- increases convergence time;
- leads to convergence to wrong solution; or
- unexpected program termination (crash).

In the case of divergence due to an unexpected termination — e.g., segfault — this is a clear indication that an error has occurred. The other possible results from faults lead to silent data corruption (SDC), but the algorithm still continues to function. SDC may also lead to longer runtimes, or to converging to the wrong solution. We rely on mathematical theory and heuristics of AMG to detect these errors as discussed below.

Checkpoint-restart is the defacto method to recover from failure by allowing the application to crash and restart from a checkpoint or beginning state. We utilize this idea in our recovery from detected errors. By observing the traversal of the AMG hierarchy, we consider each level an *implicit* checkpoint, since the data for that level is computed via the previous level in the hierarchy. It is possible to build a recovery scheme that takes advantage of this implicit checkpoint. In order to exploit this, we define a recovery function to to restart

AMG on the previous level, or if the error is deemed severe, restart the cycle. The code executed by our multilevel recovery scheme is found in Algorithm 1. If a fault is detected, we first determine its severity. Provided the fault is minor we recover from the previous level in the AMG hierarchy. Should a SDC be flagged in the same location again, or the residual check, discussed below, is triggered we rollback to our previous *explicit* checkpoint and restart the solve on the finest level.

Algorithm 1: Multilevel Restart

```

1 if retry-same-level or residual-check-failed then
2   restoreFromCheckpoint( $x^k$ )           {in-memory, fine level}
3   restartLevel( $lev_{finest}$ )
4 if down-pass then
5   if  $lev \neq lev_{finest}$  then
6     restartLevel( $lev - 1$ )
7   if  $iteration > 0$  then
8     up-pass  $\leftarrow$  TRUE
9     down-pass  $\leftarrow$  FALSE
10    restartLevel( $lev + 1$ )
11  else
12    restartLevel( $lev_{finest}$ )
13 if up-pass then
14   if  $lev \neq lev_{coarsest}$  then
15     restartLevel( $lev + 1$ )
16  else
17    up-pass  $\leftarrow$  FALSE
18    down-pass  $\leftarrow$  TRUE
19    restartLevel( $lev - 1$ )

```

In order for our restart routine to execute correctly, we create global state information comprised of direction, level, and iteration, that the solver updates as it traverses the hierarchy. Saving and restarting at some point in the hierarchy utilizes the C-language functions `sigsetjmp` to mark level or cycle restart locations and `siglongjmp` to facilitate the jumps. These functions store and restore the register state of the program at the point of the call respectively. These functions can be thought of as a label and a non-local `goto`. In addition, the solution vector periodically creates an in-memory checkpoint at the end of the V-cycle to limit the amount of roll back required. These minor augmentations are designed to be generic to allow them to be added to any sequential or parallel AMG implementation with minimal effort.

Silent Error Detectors

Faults that do not lead to an unexpected termination become SDC and go unnoticed by the standard AMG algorithm. Here, we add simple augmentations to AMG that allow the detection of SDC. These augmentations vary in their overhead, applicability to other codes, coverage, and recovery cost, as we detail below.

Residual Check

The typical stopping criterion for AMG is verifying that the relative residual is less than a given tolerance. If SDC occurs during a cycle, its effect is shown in the residual calculated for that cycle and every cycle thereafter until the error is removed. This is illustrated in our motivating example, Figure 1. Unlike other iterative linear solvers, AMG

solvers do not guarantee that the residual or relative residual monotonically decreases from cycle to cycle. Heuristics show that for a large class of problems, the residual is expected to decrease *nearly* monotonically; therefore, we devise a low cost error check that examines the newly calculated residual and compares it to the previous residual. Due to the non-monotonically decreasing nature of the residual, we use a region of plausibility for the new residual. In this paper, we consider a single order of magnitude in difference; while this is subjective, it is important to note that we are attempting to save the solver from a total restart as we detail in the next section. In addition, the scaling factor in our residual check may be modified depending on the type of problem being solved to enhance its ability to detect SDC and to prevent infinite loops. We also note that this SDC detector extends to other iterative solvers and is not specific to only AMG.

Energy Check

AMG does not guarantee that the residual decreases monotonically at the end of every cycle. However, it does guarantee a decrease in the A-norm of the error every cycle. Although we are unable to measure the error directly, we use AMG’s sense of energy to check for SDCs, termed the *energetic stability*,

$$E = \langle Ax, \mathbf{x} \rangle - 2\langle \mathbf{x}, \mathbf{b} \rangle \quad (1)$$

The energetic stability is valid on each level. That is, energy at level i in the down-pass of the V-cycle is greater than the energy calculated at level i in the down-pass in the next V-cycle. We utilize the energy check as a guarantee that the results calculated on a given level are correct before we continue to the next level. This allows recovery to proceed by the multilevel restart algorithm shown in, Algorithm 1.

In its current form, (1) is costly as it requires one sparse matrix-vector multiply (SpMV) and two inner products. The latter limits scalability at large core counts. On the down-pass and not on the coarsest level, the residual is formed. This operation provides us with $A\mathbf{x}$ at no cost. On the up-pass, the computation is more expensive, but we rewrite (1) in the following form

$$E = \langle \mathbf{r}, \mathbf{b} \rangle - 2\langle \mathbf{x} - \mathbf{b}, \mathbf{b} \rangle. \quad (2)$$

If we preform the local operations for both inner products and issue a single reduce that operates on the two local inner products in the same call, issues with scalability are minimized.

Segfault Recovery

Unlike the results of the energy or residual check where there is a global view of the result of each check once the operation completes, segfaults are local to the process on which they occur. One approach to recovery is to elevate a local segfault to a global segfault, thus allowing recovery via Algorithm 1, however, such a scheme can cause high overheads and will increase the amount of work redone. A parallel application is considered as a decomposition into two phases: communication and computation. AMG is composed of basic linear algebra operations such as SpMV operations and vector operations. All of these operations have an idempotent form at the expense of a temporary vector. For example,

$y = Ax$ produces the same result independent of the number of times the operation is executed. Due to there idempotent nature, recovery from a segfault is straightforward: restart the operation by the use of the same C functions required for Algorithm 1, but use new recovery points for the idempotent operations. The error is transient and does not manifest itself as a segfault again. If the segfault occurs during a non-idempotent operation such as communication, recovery by restart using Algorithm 1 is possible, but important state information, communication library, may be corrupted – e.g. redundantly sent messages. By message logging we are able to eliminate redundant messages, but still face possible corruption in the communication library. As the results in Section 5 show, segfaults in non-idempotent regions represent a small portion of segfaults ($< 3\%$) for the results in Table 2 and Table 3; therefore, we do not add this extra overhead to our recovery scheme.

PERFORMANCE MODEL

In this section, we construct a performance model in order to highlight the scalability of our approach and the impact on computational complexity in the scheme. In particular, we develop this model for our algorithmic based detectors and multilevel recovery scheme in order to understand the impact on the convergence properties of the AMG solver and to see the potential benefits of the methodology. In Section 5.3, we present results from fault injection experiments on solves of AMG with various error rates and detectors enabled.

Basic Model

AMG is divided into two phases: setup and solve. These phases do not overlap leading to the basic performance model, for the runtime of the solution to $Ax = b$:

$$T_{\text{total}} = T_{\text{setup}} + T_{\text{solve}} \quad (3)$$

During the solve phase of AMG, there are several choices that influence the performance of the solver such as depth of the cycle, type of cycle, smoother, and etc. In response, our performance model below (see (4)) is designed to abstract many of these choices yielding a practical tool, while at the same time providing enough detail to draw firm conclusions. T_{cycle} is the time for a single cycle (iteration) in AMG, and n_{cycles} is the number of cycles until convergence. In our model, c_i and t_i are the number of visits and time on level i , respectively. Since we consider faults only in the solve phase, we introduce a term β that represents the percent increase in iteration count due to reduced convergence as result of a SDC. β is more accurately modeled by T_{repeat} , where r_i is the number of times a level is repeated due to a SDC.

$$\begin{aligned} T_{\text{solve}} &= T_{\text{cycle}}n_{\text{cycle}}(1 + \beta) \\ &= T_{\text{cycle}}n_{\text{cycle}} + T_{\text{repeat}} \\ &= \sum_{i=0}^{n_{\text{level}}} c_i t_i n_{\text{cycle}} + \sum_{i=0}^{n_{\text{level}}} r_i t_i \end{aligned} \quad (4)$$

The number of cycles (iterations) required for AMG to converge to d digits of accuracy is given by $\frac{d}{-\log_{10}(\rho)}$, where ρ is the convergence factor, the spectral radius of the iteration matrix used during the relaxation step. If faults occur during the solve phase, convergence *potentially* deteriorates to $\hat{\rho} = \alpha\rho$

such that $\rho < \hat{\rho} < 1$. As convergence deteriorates, additional cycles are required to achieve convergence to the same accuracy. In some situations, the increase is dramatic as outlined in Figure 1. Our detection and recovery schemes limit the number of extra cycles required by maintaining the average convergence factor of the fault free problem in most cases, or a few more iterations if the average convergence factor is only slightly affected.

Segfault Recovery

To implement the segfault recovery outlined in Section 3.3, we need to issue calls to `sigsetjmp` and `siglongjmp`. These calls, along with subsequent retry of the regions after a segfault, add time not reflected in our model (4). To extend our model we modify T_{cycle} in (4) with the addition of a $T_{\text{resiliency}}$ term that is the overhead of the added resiliency measures.

$$T_{\text{resiliency}} = T_{\text{seg}} \quad (5)$$

Here T_{seg} is the overhead of segfault recovery. Recovery from a segfault has us restarting the local operation. This operation time is less than the level time. Our model provides an upper bound on the restart time by defining r_i as the sum of the maximum number of segfaults experienced on a single rank on level i during each cycle.

Residual and Energy Check

To add the residual and energy checks to our performance model in (6), we make an extension to (5), where $T_{\text{checkpoint}}$ is the time to checkpoint the solution vector on the fine level, T_{residual} is the overhead of the residual calculation, and $T_{\text{energyDown}}$ and T_{energyUp} are the overhead of doing the energy check on each level on the down and up-pass respectively. This yields

$$T_{\text{resiliency}} = T_{\text{seg}} + T_{\text{checkpoint}} + T_{\text{residual}} + T_{\text{energyDown}} + T_{\text{energyUp}}. \quad (6)$$

Comparison of Time to Solution

Faults that occur during the solve phase that require extra iterations to converge, increase the average convergence factor. As the average convergence factor approaches 1.0 the number of iterations required to converge grows exponentially. Ultimately, the same *asymptotic* convergence factor is likely achieved, yet the *effective* convergence factor for the run increases. In Figure 3, we see the relative overhead is equal to β . Each trend is associated with a different value for β , which represents the percent increase in iteration count over the fault free case. That is $\beta = 1.0$ corresponds to taking twice as many iterations to solve the problem when compared to the fault free case, or a relative overhead of 1. The overhead due to a SDC is proportional to the amount of work that is redone. For example, if a problem's average convergence factor changes from 0.2 to 0.3, this is equivalent to $\beta = 0.3$ or a 1.3x increase in the number of iterations of the original problem. This is only compounded as the average convergence factor approaches 1.0 where the number of iterations required grows exponentially. Our methods keep β small which limits extra computation.

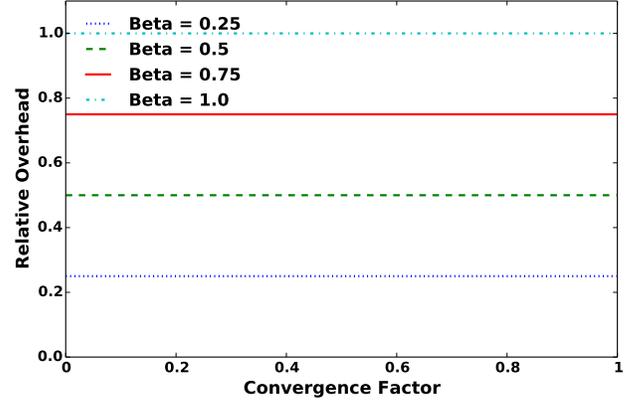


Figure 3. Relative overhead in converging to a fixed tolerance for various values of β .

EXPERIMENTAL RESULTS

To better understand AMG in a faulty environment, we use the fault injector FlipIt [4] to perturb with a bit-flip the result of a single dynamic instruction during the solve phase of the AMG solver of Hypre¹. In the following (except Section 5.1), 1000 fault injection trials are conducted on Blue Waters where we solve a 2D Laplacian with zero on the boundaries using 16 processes and 16,384 unknowns per process. Because of its rapid convergence and lightweight hierarchy, this problem highlights the overhead of our detectors and recovery scheme with more severity than a more difficult problem where the solver itself is more computationally heavy. Moreover, due to the rapid convergence for this model problem, the errors should be quickly removed by the efficient lightweight hierarchy. When a fault is injected it is classified by the fault injector in one of three main types: *pointer* refers to all calculations directly related to use of a pointers (loads, stores, and address calculation), *control* refers to all calculations of branching and control flow (comparisons for branches and modification of loop control variables), *arithmetic* refers to pure mathematical operations.

Overhead

By their nature, transient faults are infrequent events, which implies that any resiliency scheme should limit the overhead introduced in a fault free case. To determine the practicality of our detectors, it is critical to assess both the ability to detect as well as their cost (overhead). Figure 4 details the overhead with respect to solve phase execution time of each SDC detector used. We enable the multilevel recovery scheme for every configuration in which a detector is active. Because we are determining the overhead in a fault free case, the multilevel restart code is never executed; therefore, we do not include the time to restart the AMG solve in the reported times. We enable the energy check on every level to provide a worst case scenario. In addition, we in-memory checkpoint the solution vector at the end of every V-Cycle for the same reason.

¹https://computation-rnd.llnl.gov/linear_solvers/software.php

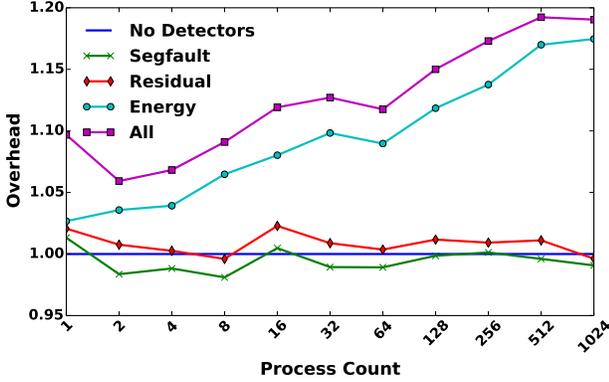


Figure 4. Overhead in solve time of fault detectors compared to the original AMG code.

We see that segfault (*Segfault*) recovery and the residual check (*Residual*) yield a lower overhead than a energy check (*Energy*). In fact, their runtimes are similar to the original code, differing by $\pm 1\%$ regardless of process count. This is offset by the fact that the energy check is more powerful, able to check for SDCs at each level. Performing the energy check on each level limits the amount of work that needs to be redone. Because of their relative low combined overhead ($< 1.5\%$), segfault recovery and the residual check, we refer to both of them together as the *Low Cost* configuration. Even with all detectors and our recovery scheme enabled in a worst case scenario, we have an observed maximal overhead of $< 20\%$. In practice, the frequency of a checkpoint, and on which levels the energy check is active would be modified to meet a more strict resiliency budget.

To better characterize the overhead of the energy check, in Figure 5 we see the percentage of level time consumed by the energy check. As we move from the finest level (0) to the coarsest level (10) the problem size decreases, but each level becomes more dense. As the density of the level increases, we see the energy check having a greater influence on level time. Although we have an increase in percentage of level time, the energy check is cheaper on the coarsest levels than on the finest level. The energy check is relatively more expensive on level 1 because although the number of unknowns in the problem is less on level 1, the number of non-zero entries is roughly the same leading to a denser matrix used in the check.

Fault Characteristics

In order to devise effective resiliency schemes, we need to determine where faults are injected into the AMG solve phase. We again investigate the worst case scenarios for both the residual and energy check. Faults are injected randomly throughout the solve phase with each dynamic instruction having a uniform probability of $1e^{-8}$. Seeding the fault injector differently for each trial yields injections in all iterations of the solve.

From Table 1 we see that most of the faults are injected into *Relaxation* and the *SpMV* routines, which includes *A* for the residual, *P*, and *R*. This is expected since most of time is spent in these routines. We classify *Other* routines as all other

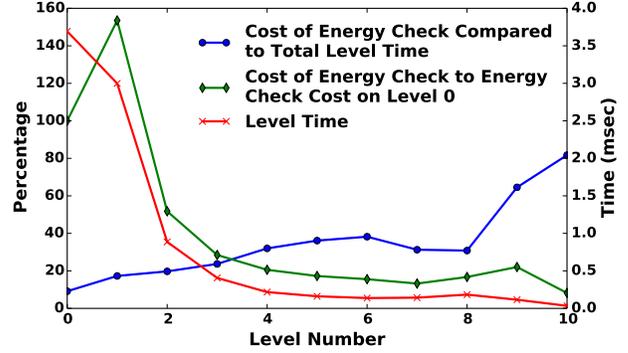


Figure 5. Characterization of cost of the energy check compared to total time on each level and cost relative to the energy check on level 0. Total level time is the sum of original level time and time for the energy check on that level.

Operation	No Detectors	Low Cost	All
Relaxation	50.1	47.4	42.5
SpMV	47	49.1	48.2
Inner product	1.1	1.7	6.7
Other	1.8	1.8	2.6

Table 1. Percentage of injections in AMG components.

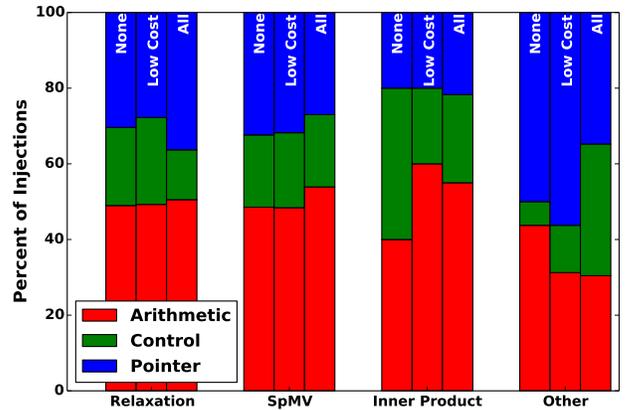


Figure 6. Breakdown of faults injected into AMG components based on the type of instruction executed.

functions used during the Hypr solve phase — e.g. cycling code, vector copy, and scaling routines. As we add the energy check that requires two inner products, we see a higher percentage of faults occurring in this routine.

Notably, in Figure 6, we see that a significant number of faults are injected into *pointer* and *arithmetic* instructions. The high degree of *pointer* injections is due to using sparse matrix data structures which uses indirection to access the data elements. A corrupted pointer often leads to a segfault, and corruption of *arithmetic* computation can produce results as shown in Figure 1, thus motivating the need for an efficient detection and recovery scheme.

Adding our SDC detectors increases resilience, but may also suffer from false positives. The comparisons for the residual and energy check represent a small portion of the dynamic

instructions during the solve phase and are unlikely to experience a SDC. The computation to form the quantities for the comparisons are more likely candidates. Regardless of where SDC occurs, a false positive in the residual or energy check is handled as if it was a true positive.

Convergence Analysis

In the following convergence study, we use the same initial guess and right hand side to limit variability in the results. Moreover, since an injected fault may lead to an increase in the number of iteration, we set the maximum number of iterations at 20. This allows 4 more iterations to converge.

Results	No Detectors	Low Cost	All
Converge	73.5	98.6	99
Did not converge	1	0.4	0
Crashed	25.5	1	1

Table 2. Percentage of trials that converge with a single injection.

In Table 2, we see that a single injection has a large impact on convergence, with 25.5% of all trials segfaulting before converging in the unmodified version of AMG. A high rate of segfaults is expected since one-third of injections are into instructions classified as *pointer*. In configurations where segfault recovery is enabled, our approach successfully recovers, thus allowing the solve to continue. With the aid of our SDC detectors we intercept 12% of injections. If uncaught, these injections would lead to extra iterations.

Results	No Detectors	Low Cost	All
Converge	0	86.2	84.8
Did not converge	0	2.8	0.2
Crashed	100	11	15

Table 3. Percentage of trials that converge with multiple injections. Average of 14 injections per trial for *Low Cost* and *All*. Average of 4 injections per trial for *No Detectors*.

With multiple injections, Table 3 shows a large deterioration in convergence for the unmodified AMG implementation with all trials crashing due to segfaults. However, our augmented versions of AMG remains convergent even in the presence of a high number of faults. Enabling the energy check increases sensitivity to SDC allowing us to detect more of them, but the check also incurs a cost. The energy check flags SDC at a rate that is 1.5x that of the residual check, but recovering from the SDC is more expensive than letting the AMG naturally iterate through the error. Some SDC that just triggers the energy check only slightly increases the energy from the previous iteration triggering the check, but not enough to have a high impact on convergence. In addition, the energy check also increases the parallel communication in routines where idempotent segfault recovery is not possible. Consequently, as the number of faults encountered per solve increases, the energy check becomes less useful.

We now look at how convergence is affected by the number of injections in both the *Low Cost* and *All* configurations. In Figure 7, we increase the average number of injections in each

trial until the probability of convergence drops below 0.5. We compute the probability of convergence by dividing the number of trials that converge by the number of trials.

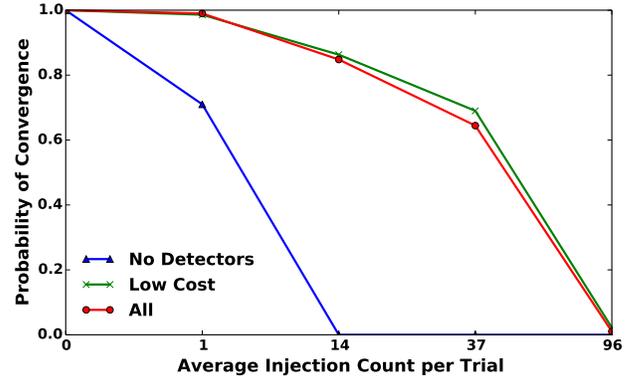


Figure 7. Convergence of AMG with multilevel recovery scheme.

Even with an average of 37 faults injected during each trial, our resilient version of AMG converges in over 69% of trials for *Low Cost* and in over 62% of trials for *All*. Again we see the duality of the energy check. It is able to detect more SDC, even those that do not significantly affect convergence. However, after this point in the testing the probability of convergence decreases dramatically. This is because the solver is making little progress due to high amounts of recovery and because the segfaults are emerging in non-idempotent routines. Indeed, 54% of trials of the *Low Cost* configuration and 75% of the *All* trials terminate due to segfaults in non-idempotent regions. This suggests that if the code were restructured with more idempotent regions, convergence rates would improve. For the remaining faults, when a SDC is detected we attempt to recover, but as we are in the recovery process we suffer more faults that require a restart allowing almost no forward progress.

CONCLUSIONS

Through the use of a combination of application specific detectors we are able to detect SDC that significantly impact convergence. With SDC detected our proposed multilevel recovery scheme is able to recover and converge with a high probability even for a high number of faults. Going forward, fault consideration is going to become a driving factor in design of long running and large scale software. With the AMG augmentations presented and evaluated in this paper, AMG has shown it is capable of being used in faulty environments. Moreover, the residual check and the local segfault recovery scheme are general and likely to be valuable in other numerical libraries.

ACKNOWLEDGMENTS

This work is sponsored by the United States Air Force Office of Scientific Research under grant FA9550-12-1-0478. This work is supported by the Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract DE-AC02-06CH11307. This research is part of the Blue Waters sustained-petascale

computing project, which is supported by the National Science Foundation (awards OCI-0725070 and ACI-1238993) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications.

REFERENCES

1. Agullo, E., Giraud, L., Guermouche, A., Roman, J., and Zounon, M. Towards resilient parallel linear Krylov solvers: recover-restart strategies. Rapport de recherche RR-8324, INRIA, July 2013.
2. Anfinson, C. J., and Luk, F. T. A linear algebraic model of algorithm-based fault tolerance. *IEEE Trans. Computers* 37, 12 (1988), 1599–1604.
3. Borkar, S. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro* 25, 6 (Nov. 2005), 10–16.
4. Calhoun, J., Olson, L., and Snir, M. FlipIt: An LLVM based fault injector for HPC. In *Proceedings of the 20th International Euro-Par Conference on Parallel Processing (Euro-Par '14)* (2014).
5. Cappello, F., Geist, A., Gropp, W. D., Kale, S., Kramer, B., and Snir, M. Toward exascale resilience: 2014 update. *Supercomputing Frontiers and Innovations 1* (2014), 1–28.
6. Casas, M., de Supinski, B. R., Bronevetsky, G., and Schulz, M. Fault resilience of the algebraic multi-grid solver. In *Proceedings of the 26th ACM international conference on Supercomputing, ICS '12*, ACM (New York, NY, USA, 2012), 91–100.
7. Chen, Z. Algorithm-based recovery for iterative methods without checkpointing. In *Proceedings of the 20th international symposium on High performance distributed computing, HPDC '11*, ACM (New York, NY, USA, 2011), 73–84.
8. Chen, Z. Online-abft: An online algorithm based fault tolerance scheme for soft error detection in iterative methods. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13*, ACM (New York, NY, USA, 2013), 167–176.
9. de Kruijf, M., Nomura, S., and Sankaralingam, K. Relax: An architectural framework for software recovery of hardware faults. In *Proceedings of the 37th International Symposium on Computer Architecture (ISCA)* (2010).
10. Elliott, J., Hoemmen, M., and Mueller, F. Evaluating the impact of SDC on the GMRES iterative solver. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium, IPDPS '14*, IEEE Computer Society (Washington, DC, USA, 2014), 1193–1202.
11. Elliott, J., Mueller, F., Stoyanov, M., and Webster, C. Quantifying the impact of single bit flips on floating point arithmetic. Tech. rep., Oak Ridge National Laboratory, August 2013.
12. Fiala, D., Mueller, F., Engelmann, C., Riesen, R., Ferreira, K., and Brightwell, R. Detection and correction of silent data corruption for large-scale high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12*, IEEE Computer Society Press (Los Alamitos, CA, USA, 2012), 78:1–78:12.
13. Hargrove, P. H., and Duell, J. C. Berkeley lab checkpoint/restart (BLCR) for linux clusters. *Journal of Physics: Conference Series* 46, 1 (2006), 494.
14. Huang, K.-H., and Abraham, J. A. Algorithm-based fault tolerance for matrix operations. *IEEE Trans. Comput.* 33, 6 (June 1984), 518–528.
15. Mishra, A., and Banerjee, P. An algorithm-based error detection scheme for the multigrid method. *IEEE Trans. Comput.* 52, 9 (Sept. 2003), 1089–1099.
16. Moody, A., Bronevetsky, G., Mohror, K., and Supinski, B. R. d. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '10*, IEEE Computer Society (Washington, DC, USA, 2010), 1–11.
17. Nicolae, B., and Cappello, F. AI-Ckpt: Leveraging memory access patterns for adaptive asynchronous incremental checkpointing. In *Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing, HPDC '13*, ACM (New York, NY, USA, 2013), 155–166.
18. Ruge, J. W., and Stüben, K. Algebraic multigrid. In *Multigrid methods*, vol. 3 of *Frontiers in Applied Mathematics*. SIAM, Philadelphia, PA, 1987, 73–130.
19. Sastry Hari, S. K., Li, M.-L., Ramachandran, P., Choi, B., and Adve, S. V. mSWAT: Low-cost hardware fault detection and diagnosis for multicore systems. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 42*, ACM (New York, NY, USA, 2009), 122–132.
20. Snir, M., Wisniewski, R. W., Abraham, J. A., Adve, S. V., Bagchi, S., Balaji, P., Belak, J., Bose, P., Cappello, F., Carlson, B., Chien, A. A., Coteus, P., DeBardleben, N. A., Diniz, P. C., Engelmann, C., Erez, M., Fazzari, S., Geist, A., Gupta, R., Johnson, F., Krishnamoorthy, S., Leyffer, S., Liberty, D., Mitra, S., Munson, T., Schreiber, R., Stearley, J., and Hensbergen, E. V. Addressing failures in exascale computing. *International Journal of High Performance Computing Applications* 28, 2 (May 2014), 127–171.
21. Sridharan, V., and Liberty, D. A study of dram failures in the field. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12*, IEEE Computer Society Press (Los Alamitos, CA, USA, 2012), 76:1–76:11.

Exploiting computing power of Xeon and Intel Xeon Phi for a Molecular Dynamics Application

Benny Mathew, Nitin Rai, Apaar Gupta and Amit Harode

Tata Consultancy Services

Quadra II, Hadapar, Pune, India 411028

{benny1.m, rai.nitin, apaar.gupta, amit.harode}@tcs.com

ABSTRACT

Molecular Dynamics (MD) is a computational technique with applicability in fields as diverse as material science, biomolecules and chemical physics. Assisted Model Building with Energy Refinement (AMBER) is an MD package and it uses Message Passing Interface (MPI) to scale in multi-core and cluster environments.

In our earlier work [1], we modified one of AMBER's algorithms called Generalized Born (GB) algorithm to run optimally on the Xeon Phi co-processor. This improved performance by 277% on the co-processor. The same changes improved performance on the host server by 80%.

In this paper, we extend our earlier work and implement a symmetric solution using both the host server and the co-processor. Since the calculations in GB algorithm involve interactions between all possible atom combinations, it has been very difficult to scale GB algorithm in distributed memory. We evaluate various alternate techniques using combination of MPI and Open Multi-Processing (OpenMP) to get a scalable solution that utilizes the computing power of both the host server as well as the co-processor.

Author Keywords

molecular dynamics; AMBER; HPC; performance; Intel Xeon Phi; nucleosome; generalized Born; parallel programming, OpenMP, MPI

ACM Classification Keywords

C.4 PERFORMANCE OF SYSTEMS

INTRODUCTION

Molecular mechanics models consist of spherical atoms connected by springs that represent bonds [2]. The internal forces experienced in the model are described using simple

mathematical functions. Hooke's law is commonly used to describe bonded interactions, and the non-bonded atoms might be treated as inelastic hard spheres or may interact according to a Lennard-Jones potential. Using these simple models, an MD simulation numerically solves Newton's equations of motion. This allows the structural changes to be observed with respect to time.

MD is used to obtain information on the time evolution of conformations of proteins and other biological macromolecules. MD is used to obtain kinetic and thermodynamic information. Simulations can provide fine detail concerning the motions of individual particles as a function of time. They can be utilized to quantify the properties of a system at a precision and on a time scale that is otherwise inaccessible.

As molecular systems consist of large number of particles with complex interaction mechanisms, it is impossible to ascertain useful properties analytically. MD simulation circumvents this problem by using numerical methods. Precise MD simulations of biological processes are computationally expensive. This is because biological processes take place in aqueous solvents and hence the solute processes as well as solvent interactions need to be simulated. The GB algorithm is an implicit solvent framework that replaces the aqueous solvent by an infinite continuum medium with electrostatic properties of solvent.

Intel's Xeon Phi coprocessor is a symmetric multiprocessor in the form factor of a PCI express device. Each coprocessor has more than 50 cores clocked at 1 GHz or more, supporting 64-bit x86 instructions. The exact number of cores depends on the model and the generation of the product. Cores of Xeon Phi coprocessors are interconnected by a high-speed bidirectional ring, which unites L2 caches of the cores into a large coherent aggregate-cache of over 25 MB in size. The coprocessor also has 6 GB to 16 GB of onboard GDDR5 memory. The speed and energy efficiency of Xeon Phi coprocessors comes from its vector units. Each core contains a vector arithmetic unit with 512-bit SIMD vectors supporting a new instruction set that includes fused multiply-add, reciprocal, square root, power and exponent operations, commonly used in physical modeling and statistical analysis. In order to completely exploit the Xeon Phi coprocessors and its host server, applications must

utilize several levels of parallelism [3]. The levels of parallelism are as follows:

- Data parallelism to employ the 512-bit vector units (vectorization)
- Task parallelism in shared memory to utilize more than 200 logical cores.
- Task parallelism in distributed memory to scale an application across multiple coprocessors or multiple compute nodes.

RELATED WORK

In this study, we work to optimize GB model implemented as part of AMBER [4] [5] simulation program to fully exploit the combined computational power of Intel Xeon host server and Xeon Phi co-processor. AMBER is a collective name for a suite of programs to carry out MD simulation.

NAMD (NAnoscale Molecular Dynamics program), GROMACS (GRoningen MAchine for Chemical Simulations), CHARMM (Chemistry at HARvard Macromolecular Mechanics) and miniMD are popular MD packages apart from AMBER. There has been significant effort to speed up MD simulation. One set of effort revolves around changing the algorithm to improve performance and the second set revolves around changes that exploit the latest available hardware.

In MD simulations, considerable time is consumed in calculating atomic interactions. One way of reducing the interactions is by ignoring remote atom interactions. This is done by constructing a list of interacting atoms within a given cut-off distance called the neighbor list. By carrying out computations for an atom with only its neighbors, the time for energy and force computations reduces without sacrificing accuracy [6].

Pennycook et al. [7] reports five times improvement in single precision Sandia's miniMD benchmark than original scalar code running on Intel Xeon Processor, using code level and algorithmic modifications. Also, addition of a single Intel Xeon Phi co-processor gives additional two times performance enhancement by using effective SIMD utilization. In their work, they use platform-specific intrinsic for force computations and building of neighbor list.

The GB algorithm's relaxation of the dielectric shielding provided by an implicit solvent model and the use of cut-off distance helps it to get better performance as compared to explicit solvent simulations. However, the performance gain is limited due to more complex functional forms and the two extra interaction stages necessary to calculate Born radii and the derivative chain rule terms contributing to the force [8]. To overcome these limitations, NAMD divides

calculations into many small work units using a three-tier decomposition scheme, assigns a balanced load of work units to processors, and schedules work units on each processor to maximize efficiency [9]. Larsson and Lindahl [8] used a rescaling transformation to make the standard GB expression a function of a single variable. This enabled an efficient tabulated solution and when this solution was implemented in GROMACS, the throughput doubled for their test cases.

Goetz et al. [10] implemented GB molecular dynamics of AMBER package on NVIDIA's graphics processing units (GPUs) by rewriting the original Fortran code into NVidia's proprietary language CUDA. Through use of unique tiled workload distribution as well by use of mixed precision model, they were able to get excellent performance at a slight cost in accuracy. Goetz's solution for AMBER offloads almost all the computation to the GPU while the host remains idle. Tanner et al. [11] implemented a solution for NAMD where there is overlap of computations on both CPU and GPU. The GB computations are carried out on the GPU and the surface area computations are carried out on the CPU.

We have optimized GB mainly for Intel Xeon Phi architecture. We use shared memory parallelism, sharing of neighbor lists and modification of loop structures to reduce communication overheads between threads. The computations retain double precision variables to preserve the accuracy of results. After optimizing the application for Xeon Phi, we implement a solution to efficiently use the computing power of both the host as well as co-processor concurrently.

GENERALIZED BORN ALGORITHM

In MD simulation, the most important quantity calculated is total energy of molecular system. From the atomic coordinates of molecules and energy, we can calculate the forces between atoms. From these forces, velocity can be calculated for each atom and using these velocities, new positions of atoms can be calculated. This flow is repeated for a number of times as required. The repetitions are usually of the order of tens of thousands. Typical biomolecules like proteins and nucleic acids hundreds to tens of thousands of atoms. In order to get meaningful results this biomolecule (solute) is immersed in an aqueous medium (solvent). Consequently, number of atoms for MD simulations increases intensely, making simulation computationally costlier. However, in bio-molecular simulations, the behavior of solute is more of interest than that of the solvent. GB is an implicit solvent model that replaces solvent medium by infinite continuum environment with similar dielectric properties of solvent, therefore eliminating calculations for aqueous medium. This approach helps to run MD simulation of large molecules for long simulation times and generate more useful result.

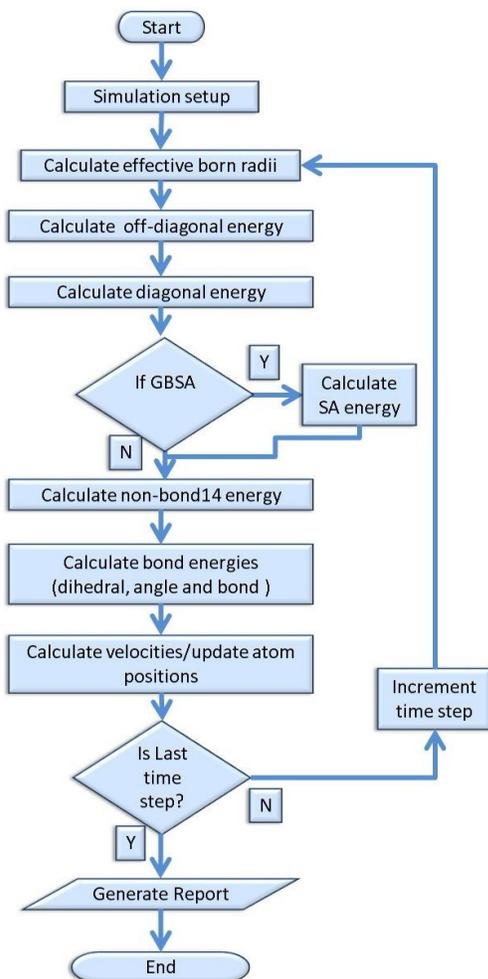


Figure 1: Generalized Born Algorithm Flow

The flow of GB simulation is shown in Figure 1. Out of the steps shown in the diagram, three steps take bulk of the simulation time and these three steps are explained in the following section.

Effective Born Radii Calculation

The effective Born radius of an atom is the degree of its burial inside the large solute molecule. The effective Born radius is calculated using relative positions of surrounding atoms. To save on computations, only atoms that are within a distance threshold from another atom are considered for calculation. However, to come up with the neighbor list, a full pass of all atoms combinations is required. Since the atoms move at the end of each time-step, neighbor list needs to be built at each time-step.

In the original GB code, each MPI [12] rank is assigned a set of atoms and Figure 2 shows some of the steps to calculate effective Born radius of each atom. There are essentially four loops in the calculation. The first loop

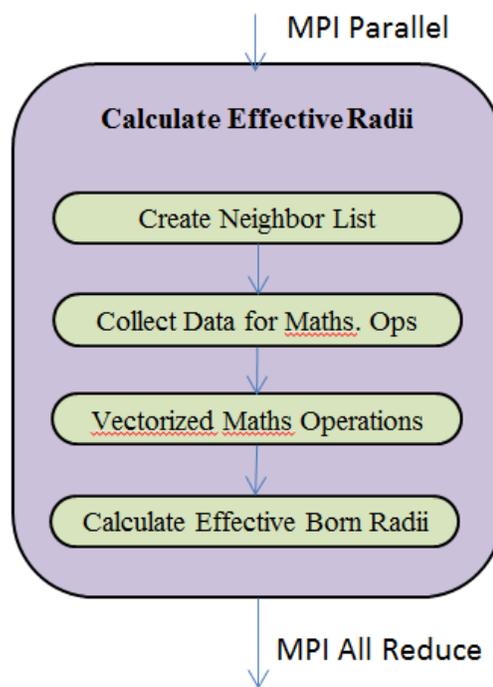


Figure 2: Effective Radii Computation

creates the neighbor list. This loop is triangular. This means that the loop starts from the atom that is next to current atom and ends with the last atom. The subsequent loops iterate only for atoms in a neighbor list. The second loop iterates to collect data on which complex mathematical operations need to be performed. By collecting the data, one can perform the operations in SIMD (Single instruction, multiple data) fashion, thereby improving performance. These mathematical operations are performed in the third loop. The mathematical operations include calculation of inverse, square root and exponents. The last loop is where effective Born radius is calculated. At the end of this loop, each node would have partial effective Born radius of several atoms. The *mpi_allreduce* call here consolidates these from each node and combines them to get final effective Born radius for each atom.

Non-diagonal Energy Calculation

The non-diagonal energy calculation has a flow similar to effective Born radius calculation. In the fourth loop instead of effective Born radii, various non-diagonal energies and forces between atoms are calculated. By looping over all pairs of atoms, it computes the gas-phase electrostatic energies, the Lennard-Jones (LJ) terms and the off-diagonal GB terms. In addition, it updates forces with derivatives of GB energy and gas-phase terms. Further, it accumulates the derivative of off-diagonal terms with respect to inverse effective radii. The threshold distance to calculate neighbor list is typically set larger than the threshold set for radii

calculation, as smaller cut off will lead to less accurate results.

Diagonal Energy Calculation

In this section of the code, the diagonal or self-energy terms of GB model is computed. It updates force vectors by taking derivatives of GB terms (including the diagonal terms) with respect to the effective radii. This calculation follows similar flow and cut-offs for neighbor list is same as that of the effective Born radii calculation. One difference in loop structure of diagonal energy calculation is that the first loop iterates through all atoms. This is unlike radii and off-diagonal calculation, where first loop is triangular.

OPTIMIZATIONS FOR INTEL XEON PHI

We use Nucleosome workload to test AMBER-GB performance on Intel Xeon Phi. Nucleosome is a compact molecular structure with 25095 atoms that acts as a basic packing unit of a usually very long DNA molecule in a living cell with nucleus. It is wound around a core of proteins, such as a thread wound on a spool. The simulation uses SHAKE algorithm with a 2-femtosecond time step and no cut-off for the non-bond terms. The cut-off used for calculation of effective Born radii is 15 Angstroms.

The performance of any MD application is measured in nanoseconds per day (ns/day). This stands for the number

of nanoseconds that the MD application will simulate if the application is run for 24 hours.

The host server used for the tests features two Intel CPUs e5-2697 with 64 GB memory. Each CPU has 12 physical cores running at 2.7 GHz. The coprocessor used is Intel Xeon Phi 7120. It has 61 cores, each running at 1.23 GHz and has 16 GB memory. The original MPI code was compiled and linked to Intel MPI library version 5 (update 2).

With the nucleosome workload, the original GB code gave throughput of 0.20 ns/day on the host. On the Xeon Phi, the throughput was 0.09 ns/day. Throughput using both host and Xeon Phi together was slower than just the host as the original code did not have any mechanism to distribute work across asymmetric nodes.

First we optimized GB to run only on Intel Xeon Phi in native mode and then optimized it to use the both the host server as well as the Xeon Phi co-processor. These optimizations are discussed as follows:

Data and Task Parallelism in Shared Memory

Various optimizations were carried out first on Xeon Phi (native mode). Only a summary of these optimizations are given here. These are explained in more detail in Harode et. al. [1].

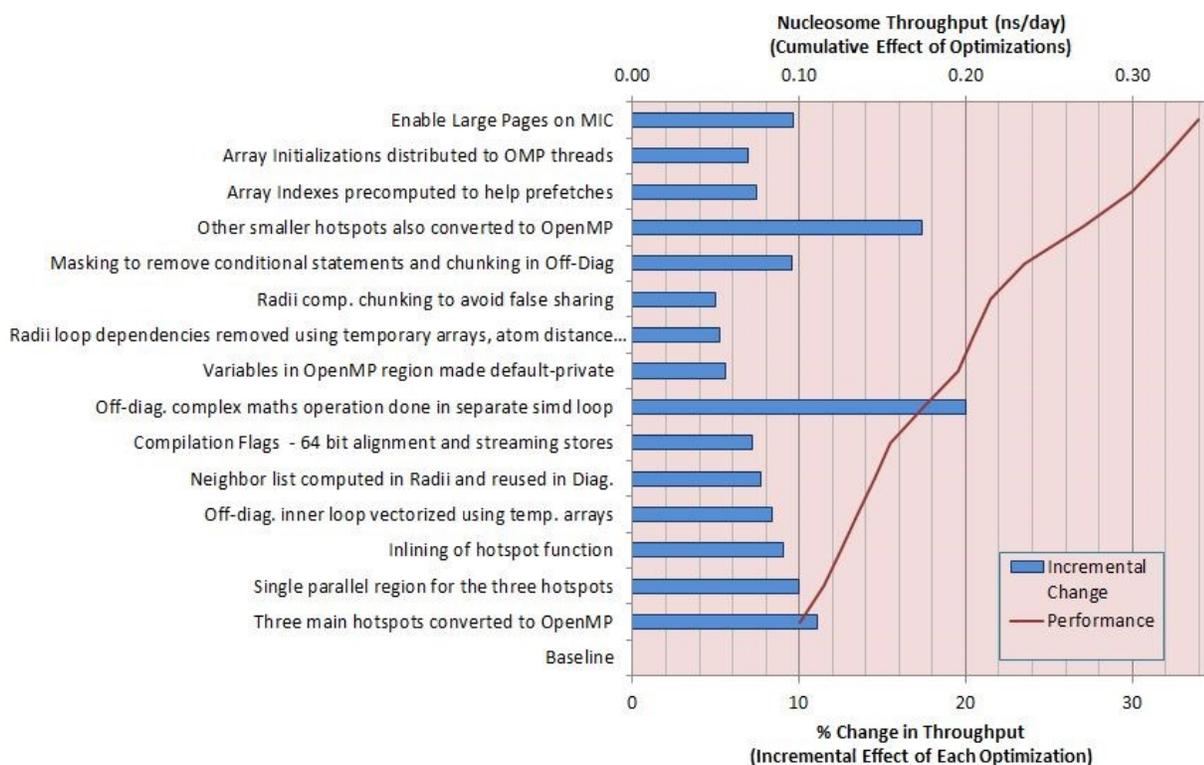


Figure 3: Native Optimizations

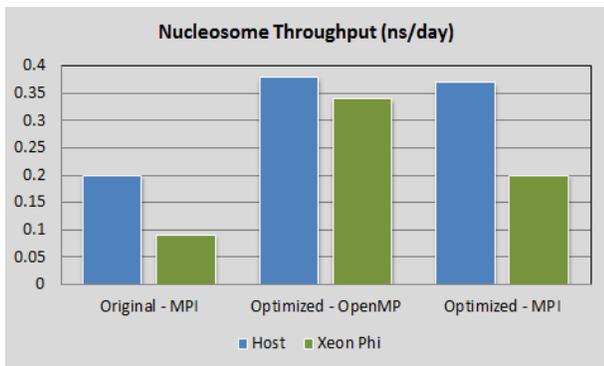


Figure 4: Performance after Native Optimizations

Though all the optimizations were manually carried out, tools like Intel’s vec-report [13] and vTune helped to identify the bottlenecks. Adding vec-report flag to Intel compiler gives detailed diagnostics about whether the loops in the code were optimized. Vec-report also gives possible strategies to get a loop to vectorize. VTune is a profiling software that gives useful information like vectorization intensity of loops, data access latencies and CPU utilization at line level as well as at function level of code.

Figure 3 shows the list of optimizations that were carried out. The line in the graphs shows the performance in ns/day as each change is combined together over the baseline. The bars show the incremental performance improvement by each optimization.

The optimizations include application of vectorization techniques like masking, latency reducing techniques like pre-computing of indexes and batching. We also employed techniques to avoid reduction and false sharing. Some of the optimizations, such as reuse of neighbor list and deferred computations to improve SIMD, required extensive code modifications.

The same set of optimizations was carried out on the original MPI code as well as on the modified one using OpenMP [14]. The version of OpenMP library used was 3.0 and it significantly improved throughput of GB algorithm on Xeon Phi. From a baseline of 0.09 ns/day on the Xeon Phi, and with the same set of optimizations, the throughput improved to 0.20 ns/day on the MPI code and to 0.34 ns/day on the OpenMP code [Figure 4]. These optimizations helped improve the throughput on the host too, though the quantum of improvement was not of the same order as observed on Xeon Phi. On the host server, for both the MPI as well as OpenMP code, the performance improved from 0.20 ns/day to 0.36 ns/day.

Task Parallelism in Distributed Memory

Now that the performance improved on both host as well as on Xeon Phi, the next step was to combine the computing power of both.

The first step was to arrive at an optimal distribution of the atoms between the host and Xeon Phi. The effective Born radii and diagonal energy computations are relatively easier to distribute. Since both effective Born radii and diagonal energy computations involve iterating through atom neighbors, the computation for each atom is more or less balanced. In addition, given that both host and Xeon Phi have the same compute power, the atoms can be equally distributed to the host and co-processor.

Diagonal energy computations are triangular, which means that more computations are required for the atoms at the start while the atoms at the end require fewer computations. Hence, the atoms cannot be equally distributed to host and Xeon Phi. Calculations proved that 70% of the atoms towards the end had the same number of computations as the 30% at the start. Hence, for diagonal energy, the computations for 70% of the atoms at the end are carried out on the Xeon Phi. Sending the atoms at the end has the added advantage that only the data of these 70% atoms need to be sent to Xeon Phi.

The next step was to choose which parallel programming technique to use on the host and Xeon Phi. To arrive at the decision, we tried the following approaches:

MPI – Symmetric

Here we use MPI for parallelism on host as well as Xeon Phi [Fig 5a]. MPI works best with one or two ranks per core. The host has 24 cores while the Intel Xeon Phi has 60. The original code distributed equal work to all MPI ranks. However, the host core is almost three times as powerful as the Xeon Phi core. Due to this asymmetric nature, the throughput was very poor at 0.1 ns/day. The code was then modified so that each rank on host processed more atoms (nearly three times) than each Xeon Phi rank. The performance improved to 0.28 ns/day [Fig 6]. However, even this figure is low considering the fact that after optimizations, the host alone is capable of giving a throughput of 0.36 ns/day.

The reason for the low performance was due to the MPI communication overhead between nodes on host and the nodes on Xeon Phi and this is discussed later in this section.

OpenMP Symmetric

In this approach [Figure 5b], we use OpenMP for parallelism on the host as well as on the Xeon Phi. The best performance was when with the use of 49 threads on the host and 180 threads on Xeon Phi. This corresponds to two threads per logical core on the host for processing and the 49th thread was used to offload computations to Xeon Phi. On the Xeon Phi there were 3 threads per core. By using the “-opt-threads-per-core=n” compiler flag, the threads per core information is passed to the compiler. This helps the

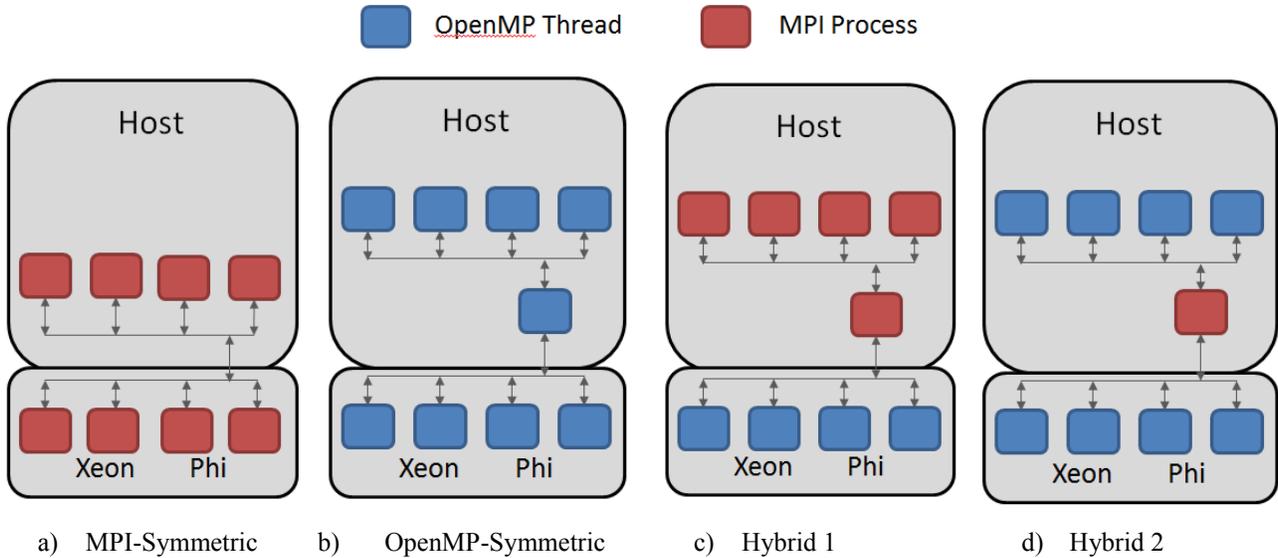


Figure 5: Different Approaches for exploiting both host and Xeon Phi Performance

compiler to perform optimizations suited to the threads to core ratio.

Using this approach, we the throughput increased to 0.58 ns/day. Significant code changes were required to implement this approach. Apart from the OpenMP changes carried out earlier, changes had to be made to facilitate optimal transfer data to and from Xeon Phi and to merge data between host and Xeon Phi.

Hybrid1

Even though OpenMP offload model gave good performance, by only employing OpenMP, we cannot scale the application across multiple servers. For multiple servers, MPI is necessary and hence we need at least one MPI rank on the host. Since the MPI implementation

performed poorly when exploiting both Xeon and Xeon Phi, a hybrid approach was needed.

In this approach, we tried using 48 MPI ranks on host (corresponding to number of logical cores) and used one of the ranks to asynchronously perform OpenMP offload to Xeon Phi [Figure 5c]. Using this approach, we got a throughput of 0.51 ns/day.

Since the performance with this hybrid approach was 12% lower than the OpenMP Symmetric solution, another variation of this approach was tried out.

Hybrid2

In this approach, there is a single MPI rank on the host. This MPI rank spawns OpenMP threads for parallelism on host. The same rank also asynchronously offloads computations to Xeon Phi [Figure 5d]. This approach gave a throughput of 0.54 ns/day. Two OpenMP threads per core were used on host and three threads per core on the Xeon Phi. This is better than the first hybrid solution and about 6% lower than the OpenMP symmetric solution.

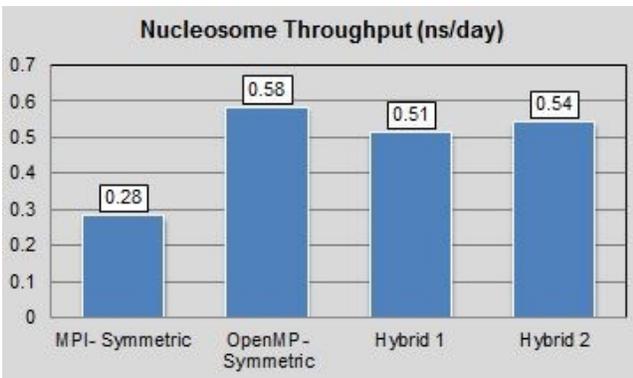


Figure 6: Performance Using Both Host and Xeon Phi

Figure 7 shows the percentage application time spent in MPI/OpenMP. These are obtained by profiling the application using ITAC [13] and vTune. The MPI symmetric approach clearly shows a very large MPI overhead on the Xeon Phi. Most of the overhead is in *MPI_Reduce_Scatter* and *MPI_AllReduce* routines.

Finally, we have two solutions, OpenMP symmetric solution suited for single host with one or more Xeon Phi co-processors and a hybrid solution that is suitable for multiple hosts.

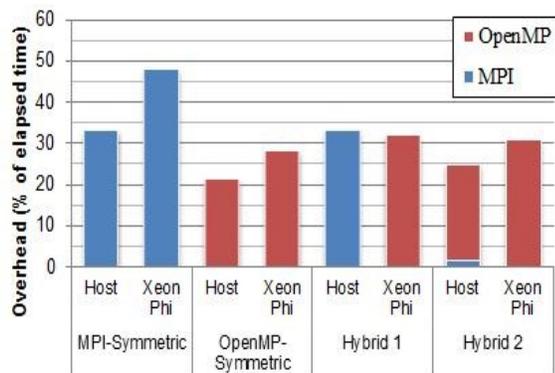


Figure 7: MPI/OpenMP Overhead

CONCLUSION AND FUTURE WORK

We optimized AMBER's GB algorithm for better performance on Xeon Phi. We achieved significant performance gain of 277% on Xeon Phi using various optimizations including the use of OpenMP. The same set of optimizations also helped the performance on the host server to improve by 80%.

We tried different models to use the computing power of host as well as Xeon Phi and achieved the best performance using an OpenMP symmetric solution. However, in order to scale with multiple servers we also implemented a hybrid model that includes MPI.

Note that the model that worked for GB algorithm is due to its unique characteristics. In the case of GB algorithm, there is lot of shared static data and hence OpenMP's shared memory model helped to reduce memory latencies. Another characteristic of GB algorithm is the fact that it is not possible to partition data perfectly between threads/ranks. Each thread computes partial values and to get the final value, there needs to be a reduction across all parallel threads/processes. Such application characteristics favor OpenMP over MPI.

Now that the GB algorithm has been optimized to run on Xeon Phi as well as to run concurrently on a host server with Xeon Phi, the next step will be to optimize it to run on multiple servers. Though the current solution is capable of running on multiple servers, the performance improvement achieved by adding a server is less than 30% of the single server throughput. In future, we plan to modify the data decomposition to achieve better scaling across servers.

ACKNOWLEDGMENTS

The authors are grateful for the support of Ashraf Bhuiyan and Michael Brown from Intel and Manoj Nambiar and Dhananjay Brahme from Tata Consultancy Services for their guidance during this project.

REFERENCES

- Harode, A., Gupta, A., Mathew B. and Rai, N. Optimization of Molecular Dynamics Application for Intel Xeon Phi Coprocessor, *International Conference on High Performance Computing and Applications, Vol 1* (in Press)
- Adcock, S.A. and McCammon, J.A. Molecular Dynamics: Survey of Methods for Simulating the Activity of Proteins, *Chemical Reviews*, Vol 106 Issue 5 (2006), 1589-1615
- Jeffers, J., and Reinders, J. *Intel Xeon Phi Coprocessor High Performance Programming*, Morgan Kaufmann (Elsevier), Feb 2013
- Case, D. A., Cheatham, T. E., Darden, T., Gohlke, H., Luo, R., Merz, K. M., Onufriev, A., Simmerling, C., Wang, B. and Woods, R. J. The AMBER Biomolecular Simulation Programs, *Journal of Computational Chemistry*, Vol. 26, Issue 16(2005), 1668-1688
- Salomon-Ferrer, R., Case, D.A. and Walker, R.C. An Overview of the AMBER Biomolecular Simulation Package, *WIREs Computational Molecular Science*, Vol. 3, Issue 2(2013), 198-210
- Rapaport, D. C. *The Art of Molecular Dynamics Simulation*, 2nd ed., Cambridge University Press, 2004.
- Pennycook, S., Hughes, C., Smelyanskiy, M. and Jarvis, S. Exploring SIMD for Molecular Dynamics, Using Intel® Xeon® Processors and Intel® Xeon Phi Coprocessors, *IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS)*, 2013, 1085-1097
- Larsson, P and Lindahl, E. A high-performance parallel-generalized Born implementation enabled by tabulated interaction *Journal of Chemical Theory and Computation*, vol. 31 (2010), 2593-60.
- Tanner, D.E., Chan, L., Phillips, J.C. and Shulten, K. Generalized Born Implicit Solvent Calculations with NAMD, *Journal of Chemical Theory and Computation*, vol 7(2011), 3635-3642
- Götz, A., Williamson, M., Xu, D., Poole, D., Grand, S. and Walker, R. Routine Microsecond Molecular Dynamics Simulations with AMBER on GPUs. 1. Generalized Born, *Journal of Chemical Theory and Computation*, vol. 8(2012), 1542-1555
- Tanner, D. E., Phillips, J. C. and Shulten, K. GPU/CPU algorithm for generalized Born/solvent-accessible surface area implicit solvent calculations, *Journal of chemical theory and computation*, vol. 8(2012), 2521-2530.
- Gropp, W., Lusk, E. L. and Skjellum, A. *Using MPI - 2nd Edition: Portable Parallel Programming with the Message Passing Interface*, The MIT Press, 1999, ISBN 978-0-2625-7132-6

13. Supalov, A., Semin, A., Klemm, M. and Dahnken, C. *Optimizing HPC Applications with Intel Cluster Tools*, Apress, 2014, ISBN 978-1-430

14. Chapman, B., Jost, G. and Pas, R.V.D. *Using OpenMP: Portable Shared Memory Parallel Programming*, The MIT Press, 2007, ISBN 978-0-2625-3302-7

Fast Parallel Conversion of Edge List to Adjacency List for Large-Scale Graphs

Shaikh Arifuzzaman^{†‡} and Maleq Khan[‡]

[†]Department of Computer Science

[‡]Network Dynamics & Simulation Science Lab, Virginia Bioinformatics Institute

Virginia Tech, Blacksburg, VA 24061, USA

{sm10, maleq}@vbi.vt.edu

ABSTRACT

In the era of Bigdata, we are deluged with massive graph data emerged from numerous social and scientific applications. In most cases, graph data are generated as lists of edges (*edge list*), where an edge denotes a link between a pair of entities. However, most of the graph algorithms work efficiently when information of the adjacent nodes (*adjacency list*) for each node is readily available. Although the conversion from edge list to adjacency list can be trivially done on the fly for small graphs, such conversion becomes challenging for the emerging large-scale graphs consisting billions of nodes and edges. These graphs do not fit into the main memory of a single computing machine and thus require distributed-memory parallel or external-memory algorithms.

In this paper, we present efficient MPI-based distributed memory parallel algorithms for converting edge lists to adjacency lists. To the best of our knowledge, this is the first work on this problem. To address the critical load balancing issue, we present a parallel load balancing scheme which improves both time and space efficiency significantly. Our fast parallel algorithm works on massive graphs, achieves very good speedups, and scales to large number of processors. The algorithm can convert an edge list of a graph with 20 billion edges to the adjacency list in less than 2 minutes using 1024 processors. Denoting the number of nodes, edges and processors by n , m , and P , respectively, the time complexity of our algorithm is $O(\frac{m}{P} + n + P)$ which provides a speedup factor of at least $\Omega(\min\{P, d_{avg}\})$, where d_{avg} is the average degree of the nodes. The algorithm has a space complexity of $O(\frac{m}{P})$, which is optimal.

Author Keywords

Parallel Algorithms; Massive Graphs; Bigdata; Edge List; Adjacency List; Load Balancing.

ACM Classification Keywords

D.1.3 Programming Techniques: Concurrent Programming—*Parallel Programming*; G.2.2 Discrete Mathematics: Graph Theory—*Graph Algorithms*

INTRODUCTION

Graph (network) is a powerful abstraction for representing underlying relations in large unstructured datasets. Examples include the web graph [11], various social networks, e.g., Facebook, Twitter [17], collaboration networks [19], infrastructure networks (e.g., transportation networks, telephone networks) and biological networks [16].

We denote a graph by $G(V, E)$, where V and E are the set of vertices (nodes) and edges, respectively, with $m = |E|$ edges and $n = |V|$ vertices. In many cases, a graph is specified by simply listing the edges $(u, v), (v, w), \dots \in E$, in an arbitrary order, which is called *edge list*. A graph can also be specified by a collection of adjacency lists of the nodes, where the *adjacency list* of a node v is the list of nodes that are adjacent to v . Many important graph algorithms, such as computing shortest path, breadth-first search, and depth-first search are executed by exploring the neighbors (adjacent nodes) of the nodes in the graph. As a result, these algorithms work efficiently when the input graph is given as adjacency lists. Although both edge list and adjacency list have a space requirement of $O(m)$, scanning all neighbors of node v in an edge list can take as much as $O(m)$ time compared to $O(d_v)$ time in adjacency list, where d_v is the degree of node v .

Adjacency matrix is another data structure used for graphs. Much of the earlier work [3, 15] use adjacency matrix $A[.,.]$ of order $n \times n$ for a graph with n nodes. Element $A[i, j]$ denotes whether node j is adjacent to node i . All adjacent nodes of i can be determined by scanning the i -th row, which takes $O(n)$ time compared to $O(d_i)$ time for adjacency list. Further, adjacency matrix has a prohibitive space requirement of $O(n^2)$ compared to $O(m)$ of adjacency list. In a real-world network, m can be much smaller than n^2 as the average degree of a node can be significantly smaller than n . Thus adjacency matrix is not suitable for the analysis of emerging large-scale networks in the age of bigdata.

In most cases, graphs are generated as list of edges since it is easier to capture pairwise interactions among entities in a system in arbitrary order than to capture all interactions of a single entity at the same time. Examples include captur-

ing person-person connection in social networks and protein-protein links in protein interaction networks. This is true even for generating massive random graphs [2, 14] which is useful for modeling very large system. As discussed by Leskovec et. al [18], some patterns only exist in massive datasets and they are fundamentally different from those in smaller datasets. While generating such massive random graphs, algorithms usually output edges one by one. Edges incident on a node v are not necessarily generated consecutively. Thus a conversion of edge list to adjacency list is necessary for analyzing these graphs efficiently.

Why do we need parallel algorithms? With unprecedented advancement of computing and data technology, we are deluged with massive data from diverse areas such as business and finance [5], computational biology [12] and social science [7]. The web has over 1 trillion webpages. Most of the social networks, such as, Twitter, Facebook, and MSN, have millions to billions of users [13]. These networks hardly fit in the memory of a single machine and thus require external memory or distributed memory parallel algorithms. Now external memory algorithms can be very I/O intensive leading to a large runtime. Efficient distributed memory parallel algorithms can solve both problems (runtime and space) by distributing computing tasks and data to multiple processors.

In a sequential settings with the graphs being small enough to be stored in the main memory, the problem of converting an edge list to adjacency list is trivial as described in the next section. However, the problem in a distributed-memory setting with massive graphs poses many non-trivial challenges. The neighbors of a particular node v might reside in multiple processors which need to be combined efficiently. Further, computation loads must be well-balanced among the processors to achieve a good performance of the parallel algorithm. Like many others, this problem demonstrates how a simple trivial problem can turn into a challenging problem when we are dealing with bigdata.

Contributions. In this paper, we study the problem of converting *edge list* to *adjacency list* for large-scale graphs. We present MPI-based distributed-memory parallel algorithms which work for both directed and undirected graphs. We devise a parallel load balancing scheme which balances the computation load very well and improves the efficiency of the algorithms significantly, both in terms of runtime and space requirement. Furthermore, we present two efficient merging schemes for combining neighbors of a node from different processors— *message-based* and *external-memory* merging— which offer a convenient trade-off between space and runtime. Our algorithms work on massive graphs, demonstrate very good speedups on both real and artificial graphs, and scale to a large number of processors. The edge list of a graph with $20B$ edges can be converted to adjacency list in two minutes using 1024 processors. We also provide rigorous theoretical analysis of the time and space complexity of our algorithms. The time and space complexity of our algorithms are $O(\frac{m}{P} + n + P)$ and $O(\frac{m}{P})$, respectively, where n , m , and P are the number of the nodes, edges, and processors,

respectively. The speedup factor is at least $\Omega(\min\{P, d_{avg}\})$, where d_{avg} is the average degree of the nodes.

Organization. The rest of the paper is organized as follows. The preliminary concepts and background of the work are briefly described in *preliminaries and background* section. Our parallel algorithm along with the load balancing scheme is presented in *parallel algorithm* section. In section *dataset and experimental setup*, we present our dataset and specification of computing resources used for experiments. We discuss our experimental results in section *performance analysis*, and we conclude thereafter.

PRELIMINARIES AND BACKGROUND

In this section, we describe the basic definitions used in this paper and then present a sequential algorithm for converting edge list to adjacency list.

Basic definitions

We assume n vertices of the graph $G(V, E)$ are labeled as $0, 1, 2, \dots, n - 1$. We use the words *node* and *vertex* interchangeably. If $(u, v) \in E$, we say u and v are neighbors of each other. The set of all adjacent nodes (neighbors) of $v \in V$ is denoted by N_v , i.e., $N_v = \{u \in V | (u, v) \in E\}$. The degree of v is $d_v = |N_v|$.

In edge-list representation, edges $(u, v) \in E$ are listed one after another without any particular order. Edges incident to a particular node v are not necessarily listed together. On the other hand, in adjacency-list representation, for all v , adjacent nodes of v , N_v , are listed together. An example of these representations is shown in Figure 1.

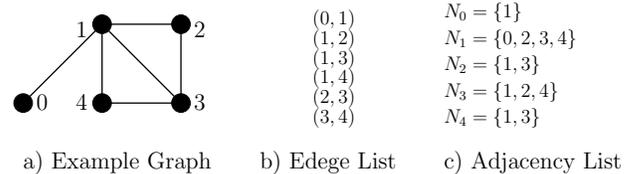


Figure 1: The edge list and adjacency list representations of an example graph with 5 nodes and 6 edges.

A Sequential Algorithm

The sequential algorithm for converting edge list to adjacency list works as follows. Create an empty list N_v for each node v , and then, for each edge $(u, v) \in E$, include u in N_v and v in N_u . The pseudocode of the sequential algorithm is given in Figure 2. For a directed graph, line 5 of the algorithm should be omitted since a directed edge (u, v) doesn't imply that there is also an edge (v, u) . In our subsequent discussion, we assume that the graph is undirected. However, the algorithm also works for the directed graph with the mentioned modification.

This sequential algorithm is optimal since it takes $O(m)$ time to process $O(m)$ edges and thus can not be further improved. The algorithm has a space complexity of $O(m)$.

```

1: for each  $v \in V$  do
2:    $N_v \leftarrow \emptyset$ 
3: for each  $(u, v) \in E$  do
4:    $N_u \leftarrow N_u \cup \{v\}$ 
5:    $N_v \leftarrow N_v \cup \{u\}$ 

```

Figure 2: Sequential algorithm for converting edge list to adjacency list.

For small graphs that can be stored wholly in the main memory, the conversion in a sequential setting is trivial. However, emerging massive graphs pose many non-trivial challenges in terms of memory and execution efficiency. Such graphs might not fit in the local memory of a single computing node. Even if some of them fit in the main memory, the runtime might be prohibitively large. Efficient parallel algorithms can solve this problem by distributing computation and data among computing nodes. We present our parallel algorithm in the next section.

THE PARALLEL ALGORITHM

First we present the computational model and an overview of our parallel algorithm. A detailed description follows thereafter.

Computation Model

We develop parallel algorithms for message passing interface (MPI) based distributed-memory parallel systems, where each processor has its own local memory. The processors do not have any shared memory, one processor cannot directly access the local memory of another processor, and the processors communicate via exchanging messages using MPI constructs.

Overview of the Algorithm

Let P be the number of processor used in the computation and E be the list of edges given as input. Our algorithm has two phases of computation. In Phase 1, the edges E are partitioned into P initial partitions E_i , and each processor is assigned one such partition. Each processor then constructs neighbor lists from the edges of its own partition. However, edges incident to a particular node might reside in multiple processors, which creates multiple partial adjacency lists for the same node. In Phase 2 of our algorithms, such adjacency lists are merged together. Now, performing Phase 2 of the algorithm in a cost-effective way is very challenging. Further, computing loads among processors in both phases need to be balanced to achieve a significant runtime efficiency. The load balancing scheme should also make sure that space requirement among processors are also balanced so that large graphs can be processed. We describe the phases of our parallel algorithm in detail as follows.

(Phase 1) Local Processing

The algorithm partitions the set of edges E into P partitions E_i such that $E_i \subseteq E$, $\bigcup_k E_k = E$ for $0 \leq k \leq P - 1$. Each partition E_i has almost $\frac{m}{P}$ edges— to be exact, $\lceil \frac{m}{P} \rceil$ edges, except for the last partition which has slightly fewer ($m -$

$(p - 1) \lceil \frac{m}{P} \rceil$). Processor i is assigned partition E_i . Processor i then constructs adjacency lists N_v^i for all nodes v such that $(\cdot, v) \in E_i$ or $(v, \cdot) \in E_i$. Note that adjacency list N_v^i is only a partial adjacency list since other partitions E_j might have edges incident on v . We call N_v^i local adjacency list of v in partition i . The pseudocode for Phase 1 computation is presented in Figure 3.

```

1: Each processor  $i$ , in parallel, executes the following.
2: for  $(u, v) \in E_i$  do
3:    $N_v^i \leftarrow N_v^i \cup \{u\}$ 
4:    $N_u^i \leftarrow N_u^i \cup \{v\}$ 

```

Figure 3: Algorithm for performing Phase 1 computation.

This phase of computation has both the runtime and space complexity of $O(\frac{m}{P})$ as shown in Lemma 1 .

LEMMA 1. *Phase 1 of our parallel algorithm has both the runtime and space complexity of $O(\frac{m}{P})$.*

Proof: Each initial partition i has $|E_i| = O(\frac{m}{P})$ edges. Executing Line 3-4 in Figure 3 for $O(\frac{m}{P})$ edges requires $O(\frac{m}{P})$ time. Now the total space required for storing local adjacency lists N_v^i in partition i is $2|E_i| = O(\frac{m}{P})$. \square

Thus the computing loads and space requirements in Phase 1 are well-balanced. The second phase of our algorithm constructs the final adjacency list N_v from local adjacency lists N_v^i from all processors i . Note that balancing load for Phase 1 doesn't make load well balanced for Phase 2 which requires a more involved load balancing scheme as described later in the following sections.

(Phase 2) Merging Local Adjacency Lists

Once all processors complete constructing local adjacency lists N_v^i , final adjacency lists N_v are created by merging N_v^i from all processors i as follows.

$$N_v = \bigcup_{i=0}^{P-1} N_v^i \quad (1)$$

The scheme used for merging local adjacency lists has significant impact on the performance of the algorithm. One might think of using a dedicated *merger* processor. For each node $v \in V_i$, the merger collects N_v^i from all other processors and merges them into N_v . This requires $O(d_v)$ time for node v . Thus the runtime complexity for merging adjacency lists of all $v \in V$ is $O(\sum_{v \in V} d_v) = O(m)$, which is at most as good as the sequential algorithm.

Next we present our efficient parallel merging scheme which employs P parallel mergers.

An Efficient Parallel Merging Scheme

To parallelize Phase 2 efficiently, our algorithm distributes the corresponding computation disjointly among processors. Each processor i is responsible for merging adjacency lists N_v for nodes v in $V_i \subset V$ such that for any i and j , $V_i \cap V_j = \emptyset$ and $\bigcup_i V_i = V$. Note that this partitioning of

nodes is different from the initial partitioning of edges. How the nodes in V is distributed among processors crucially affects the load balancing and performance of the algorithm. Further, this partitioning and load balancing scheme should be parallel to ensure the efficiency of the algorithm. Later in this section, we discuss a parallel algorithm to partition set of nodes V which makes both space requirement and runtime well-balanced. Once the partitions V_i are given, the scheme for parallel merging works as follows.

- *Step 1:* Let S_i be the set of all local adjacency lists in partition i . Processor i divides S_i into P disjoint subsets S_i^j , $0 \leq j \leq P - 1$, as defined below.

$$S_i^j = \{N_v^i : v \in V_j\}. \quad (2)$$

- *Step 2:* Processor i sends S_i^j to all other processors j . This step introduces non-trivial efficiency issues which we shall discuss shortly.
- *Step 3:* Once processor i gets S_j^i from all processors j , it constructs N_v for all $v \in V_i$ by the following equation.

$$N_v = \bigcup_{k: N_v^k \in S_k^i} N_v^k \quad (3)$$

We present two methods for performing Step 2 of the above scheme. The first method explicitly exchanges messages among processors to send and receive S_i^j by using message buffers (main memory). The other method uses disk space (external memory) to exchange S_i^j . We call the first method *message-based merging* and the second *external-memory merging*.

(1) Message-based Merging: Each processor i sends S_i^j directly to processor j via messages. Specifically, processor i sends $|N_v^i|$ (with a message $\langle v, N_v^i \rangle$) to processor j where $v \in V_j$. A processor might send multiple lists to another processor. In such cases, messages to a particular processor are bundled together to reduce communication overhead. Once a processor i receives messages $\langle v, N_v^j \rangle$ from other processors, for $v \in V_i$, it computes $N_v = \bigcup_{j=0}^{P-1} N_v^j$. The pseudocode of this algorithm is given in Figure 4.

```

1: for each  $v$  s.t.  $(\cdot, v) \in E_i \vee (v, \cdot) \in E_i$  do
2:   Send  $\langle v, N_v^i \rangle$  to proc.  $j$  where  $v \in V_j$ 
3: for each  $v \in V_i$  do
4:    $N_v \leftarrow \emptyset$ 
5: for each  $\langle v, N_v^j \rangle$  received from any proc.  $j$  do
6:    $N_v \leftarrow N_v \cup N_v^j$ 

```

Figure 4: Parallel algorithm for merging local adjacency lists to construct final adjacency lists N_v . A message, denoted by $\langle v, N_v^i \rangle$, refers to local adjacency lists of v in processor i .

(2) External-memory Merging: Each processor i writes S_i^j in intermediate disk files F_i^j , one for each processor j . Processor i reads all files F_j^i for partial adjacency lists N_v^j for each $v \in V_i$ and merges them to final adjacency lists using step 3 of the above scheme. However, processor i doesn't

read in the whole file into its main memory. It only stores local adjacency lists N_v^j of a node v at a time, merges it to N_v , releases memory and then proceeds to merge the next node $v + 1$. This works correctly since while writing S_i^j in F_i^j , local adjacency lists N_v^i are listed in the sorted order of v . External-memory merging thus has a space requirement of $O(\max_v d_v)$. However, the I/O operation leads to a higher runtime with this method than message-based merging, although the asymptotic runtime complexity remains the same. We demonstrate this space-runtime tradeoff between these two methods in our *performance analysis* section.

The runtime and space complexity of parallel merging depends on the partitioning of V . Next, we discuss the partitioning and load balancing scheme followed by the complexity analyses.

Partitioning and Load Balancing

The performance of the algorithm depends on how loads are distributed. In Phase 1, distributing the edges of the input graph evenly among processors provides an even load balancing both in terms of runtime and space leading to both space and runtime complexity of $O(\frac{m}{P})$. However, Phase 2 is computationally different than Phase 1 and requires different partitioning and load balancing scheme.

In Phase 2 of our algorithm, set of nodes V is divided into P subsets V_i where processor i merges adjacency lists N_v for all $v \in V_i$. The time for merging N_v of a node v (referred to as *merging cost* henceforth) is proportional to the degree $d_v = |N_v|$ of node v . Total cost for merging incurred on a processor i is $\Theta(\sum_{v \in V_i} d_v)$. Distributing equal number of nodes among processors may not make the computing load well-balanced in many cases. Some nodes may have large degrees and some very small. As shown in Figure 5, distribution of merging cost ($\sum_{v \in V_i} d_v$) across processors is very uneven with an equal number of nodes assigned to each processor. Thus the set V should be partitioned in such a way that the cost of merging is almost equal in all processors.

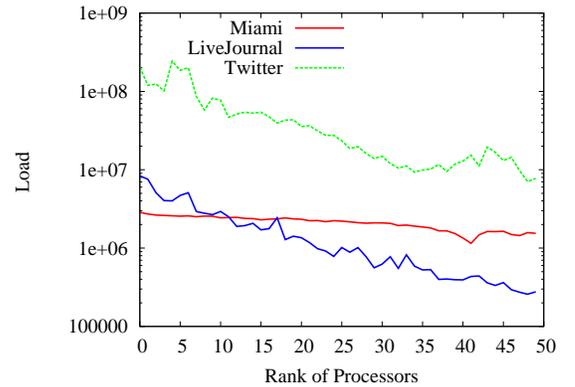


Figure 5: Load distribution among processors for LiveJournal, Miami and Twitter before applying the load balancing scheme. Summary of our dataset is provided in Table 2.

Let, $f(v)$ be the cost associated with constructing N_v by receiving and merging local adjacency lists for a node $v \in V$.

We need to compute P disjoint partitions of node set V such that for each partition V_i ,

$$\sum_{v \in V_i} f(v) \approx \frac{1}{P} \sum_{v \in V} f(v).$$

Now, note that, for each node v , total size of the local adjacency lists received (in Line 5 in Figure 4) equals the number of adjacent nodes of v , i.e., $|N_v| = d_v$. Merging local adjacency lists N_v^j via set union operation (Line 6) also requires d_v time. Thus, $f(v) = d_v$. Now, since the adjacent nodes of a node v can reside in multiple processors, computing $f(v) = |N_v| = d_v$ requires communication among multiple processors. For all v , computing $f(v)$ sequentially requires $O(m + n)$ time which diminishes the advantages gained by the parallel algorithm. Thus, we compute $f(v) = d_v$ for all v in parallel, in $O(\frac{n+m}{P} + c)$ time, where c is the communication cost. We will discuss the complexity shortly. This algorithm works as follows: for determining d_v for $v \in V$ in parallel, each processor i computes d_v for $\frac{n}{P}$ nodes v , where v starts from $\frac{in}{P}$ to $\frac{(i+1)n}{P} - 1$. Such nodes v satisfy the equation, $\lfloor \frac{v}{n/P} \rfloor = i$. Now, for each local adjacency list N_v^i constructed in Phase 1, processor i sends $d_v^i = |N_v^i|$ to processor $j = \frac{v}{n/P}$ with a message $\langle v, d_v^i \rangle$. Once processor i receives messages $\langle v, d_v^j \rangle$ from other processors, it computes $f(v) = d_v = \sum_{j=0}^{P-1} d_v^j$ for all nodes v such that $\lfloor \frac{v}{n/P} \rfloor = i$. The pseudocode of the parallel algorithm for computing $f(v) = d_v$ is given in Figure 6.

```

1: for each  $v$  s.t.  $(\cdot, v) \in E_i \vee (v, \cdot) \in E_i$  do
2:    $d_v^i \leftarrow |N_v^i|$ 
3:    $j \leftarrow \frac{v}{n/P}$ 
4:   Send  $\langle v, d_v^i \rangle$  to processor  $j$ 
5: for each  $v$  s.t.  $\lfloor \frac{v}{n/P} \rfloor = i$  do
6:    $d_v \leftarrow 0$ 
7:   for each  $\langle v, d_v^j \rangle$  received from any proc.  $j$  do
8:      $d_v \leftarrow d_v + d_v^j$ 

```

Figure 6: Parallel algorithm executed by each processor i for computing $f(v) = d_v$.

Once $f(v)$ is computed for all $v \in V$, we compute cumulative sum $F(t) = \sum_{v=0}^t f(v)$ in parallel by using a parallel prefix sum algorithm [4]. Each processor i computes and stores $F(t)$ for nodes t , where t starts from $\frac{in}{P}$ to $\frac{(i+1)n}{P} - 1$. This computation takes $O(\frac{n}{P} + P)$ time. Then, we need to compute V_i such that computation loads are well-balanced among processors. Partitions V_i are disjoint subset of consecutive nodes, i.e., $V_i = \{n_i, n_i + 1, \dots, n_{(i+1)} - 1\}$ for some node n_i . We call n_i *start node* or *boundary node* of partition i . Now, V_i is computed in such a way that the sum $\sum_{v \in V_i} f(v)$ becomes almost equal ($\frac{1}{P} \sum_{v \in V} f(v)$) for all partitions i . At the end of this execution, each processor i knows n_i and $n_{(i+1)}$. Algorithm presented in [6] compute V_i for the problem of triangle counting. The algorithm can also be applied for our problem

to compute V_i using cost function $f(v) = d_v$. In summary, computing load balancing for Phase 2 has the following main steps.

- *Step 1*: Compute cost $f(v) = d_v$ for all v in parallel by the algorithm shown in Figure 6.
- *Step 2*: Compute cumulative sum $F(v)$ by a parallel prefix sum algorithm [4].
- *Step 3*: Compute boundary nodes n_i for every subset $V_i = \{n_i, \dots, n_{(i+1)} - 1\}$ using the algorithms [1, 6].

LEMMA 2. *The algorithm for balancing loads for Phase 2 has a runtime complexity of $O(\frac{n+m}{P} + P + \max_i M_i)$ and a space requirement of $O(\frac{n}{P})$, where M_i is the number of messages received by processor i in Step 1.*

Proof: For *Step 1* of the above load balancing scheme, executing Line 1-4 (Figure 6) requires $O(|E_i|) = O(\frac{m}{P})$ time. The cost for executing Line 5-6 is $O(\frac{n}{P})$ since there are $\frac{n}{P}$ nodes v such that $\lfloor \frac{v}{n/P} \rfloor = i$. Each processor i sends a total of $O(\frac{m}{P})$ messages since $|E_i| = \frac{m}{P}$. If the number of messages received by processor i is M_i , then Line 7-8 of the algorithm has a complexity of $O(M_i)$ (we compute bounds for M_i in Lemma 3). Computing *Step 2* has a computational cost of $O(\frac{n}{P} + P)$ [4]. *Step 3* of the load balancing scheme requires $O(\frac{n}{P} + P)$ time [1, 6]. Thus the runtime complexity of the load balancing scheme is $O(\frac{n+m}{P} + P + \max_i M_i)$. Storing $f(v)$ for $\frac{n}{P}$ nodes has a space requirement of $O(\frac{n}{P})$. \square

LEMMA 3. *Number of messages M_i received by processor i in Step 1 of load balancing scheme is bounded by $O(\min\{n, \sum_{in/P}^{(i+1)n/P-1} d_v\})$.*

Proof: Referring to Figure 6, each processor i computes d_v for $\frac{n}{P}$ nodes v , where v starts from $\frac{in}{P}$ to $\frac{(i+1)n}{P} - 1$. For each v , processor i may receive messages from at most $(P - 1)$ other processors. Thus, the number of received messages is at most $\frac{n}{P} \times (p-1) = O(n)$. Now, notice that, when all neighbors $u \in N_v$ of v reside in different partitions E_j , processor i might receive as much as $|N_v| = d_v$ messages for node v . This gives another upper bound, $M_i = O(\sum_{in/P}^{(i+1)n/P-1} d_v)$. Thus we have $M_i = O(\min\{n, \sum_{in/P}^{(i+1)n/P-1} d_v\})$. \square

In most of the practical cases, each processor receives much smaller number of messages than that specified by the theoretical upper bound. Now, for each node v , processor i receives messages actually from fewer than $P - 1$ processors. Let, for node v , processor i receives messages from $O(P.l_v)$ processors, where l_v is a real number ($0 \leq l_v \leq 1$). Thus total number of message received, $M_i = O(\sum_{in/P}^{(i+1)n/P-1} P.l_v)$. To get a crude estimate of M_i , let $l_v = l$ for all v . The term l can be thought of as the average over all l_v . Then $M_i = O(\frac{n}{P} \cdot P \cdot l) = O(n \cdot l)$. As shown in Table 1, the actual number of messages received M_i is up to $7 \times$ smaller than the theoretical bound.

LEMMA 4. *Using the load balancing scheme discussed in this section, Phase 2 of our parallel algorithm has a runtime*

Network	n	$\sum_{\frac{in}{P}-1}^{(i+1)n} d_v$	M_i	$l(\text{avg.})$
Miami	2.1M	2.17M	600K	0.27
LiveJournal	4.8M	2.4M	560K	0.14
PA(5M, 20)	5M	2.48M	1.4M	0.28

Table 1: Number of messages received in practice compared to the theoretical bounds. This results report $\max_i M_i$ with $P = 50$. Summary of our dataset is provided in Table 2.

complexity of $O(\frac{m}{P})$. Further, the space required to construct all final adjacency lists N_v in a partition is $O(\frac{m}{P})$.

Proof: Line 1-2 in the algorithm shown in Figure 4 requires $O(|E_i|) = O(\frac{m}{P})$ time for sending at most $|E_i|$ edges to other processors. Now, with load balancing, each processor receives and merges at most $O(\sum_{v \in V} d_v / P) = O(\frac{m}{P})$ edges (Line 5-6). Thus the cost for merging local lists N_v^j into final list N_v has a runtime of $O(\frac{m}{P})$. Since the total size of the local and final adjacent lists in a partition is $O(\frac{m}{P})$, the space requirement is $O(\frac{m}{P})$. \square

The runtime and space complexity of our complete parallel algorithm are formally presented in Theorem 1.

THEOREM 1. *The runtime and space complexity of our parallel algorithm is $O(\frac{m}{P} + P + n)$ and $O(\frac{m}{P})$, respectively.*

Proof: The proof follows directly from Lemma 1, 2, 3, and 4. \square

The total space required by all processors to process m edges is $O(m)$. Thus the space complexity $O(\frac{m}{P})$ of our parallel algorithm is optimal.

Performance gain with load balancing: Cost for merging incurred on each processor i is $\Theta(\sum_{v \in V_i} d_v)$ (Figure 4). Without load balancing, this cost $\Theta(\sum_{v \in V_i} d_v)$ can be as much as $\Theta(m)$ (it is easy to construct such skewed graphs) leading the runtime complexity of the algorithm $\Theta(m)$. With load balancing scheme our algorithm achieves a runtime of $O(\frac{m}{P} + P + n) = O(\frac{m}{P} + \frac{m}{d_{avg}})$, for usual case $n > P$. Thus, by simple algebraic manipulation, it is easy to see, the algorithm with load balancing scheme achieves a $\Omega(\min\{P, d_{avg}\})$ -factor gain in runtime efficiency over the algorithm without load balancing scheme. In other words, the algorithm gains a $\Omega(P)$ -fold improvement in speedup when $d_{avg} \geq P$ and $\Omega(d_{avg})$ -fold otherwise. We demonstrate this gain in speedup with experimental results in our *performance analysis* section.

DATA AND EXPERIMENTAL SETUP

We present the datasets and the specification of computing resources used in our experiments below.

Datasets. We used both real-world and artificially generated networks for evaluating the performance of our algorithm. A summary of all the networks is provided in Table 2. The number of nodes in our networks ranges from 37K to 1B and the number of edges from 0.36M to 20B. Twitter [17], web-BerkStan, Email-Enron and LiveJournal [20] are real-

Network	Nodes	Edges	Source
Email-Enron	37K	0.36M	SNAP [20]
web-BerkStan	0.69M	13M	SNAP [20]
Miami	2.1M	100M	[9]
LiveJournal	4.8M	86M	SNAP [20]
Twitter	42M	2.4B	[17]
Gnp(n, d)	n	$\frac{1}{2}nd$	Erdős-Rényi
PA(n, d)	n	$\frac{1}{2}nd$	Pref. Attachment

Table 2: Dataset used in our experiments. Notations K, M and B denote thousands, millions and billions, respectively.

world networks. Miami is a synthetic, but realistic, social contact network [6, 9] for Miami city. Networks Gnp(n, d) and PA(n, d) are random networks. Gnp(n, d) is generated using the Erdős-Rényi random graph model [10] with n nodes and d average degree. Network PA(n, d) is generated using preferential attachment (PA) model [8] with n nodes and average degree d . Both the real-world and PA(n, d) networks have skewed degree distributions which make load balancing a challenging task and give us a chance to measure the performance of our algorithms in some of the worst case scenarios.

Experimental Setup. We perform our experiments using a high performance computing cluster with 64 computing nodes, 16 processors (Sandy Bridge E5-2670, 2.6GHz) per node, memory 4GB/processor, and operating system SLES 11.1.

PERFORMANCE ANALYSIS

In this section, we present the experimental results evaluating the performance of our algorithm.

Load distribution

Load distribution among processors can be very uneven without applying our load balancing scheme, as discussed in *partitioning and load balancing* section. We show a comparison of load distribution on various networks with and without load balancing scheme in Figure 7. Our scheme provides an almost equal load among the processors, even for graphs with very skewed degree distribution such as LiveJournal and Twitter. Loads are significantly uneven for such skewed networks without load balancing scheme.

Strong Scaling

Figure 8 shows strong scaling (speedup) of our algorithm on LiveJournal, Miami and Twitter networks with and without load balancing scheme. Our algorithm demonstrates very good speedups, e.g., it achieves a speedup factor of ≈ 300 with 1024 processors for Twitter network. Speedup factors increase almost linearly for all networks, and the algorithm scales to a large number of processors. Figure 8 also shows the speedup factors the algorithm achieves without load balancing scheme. Speedup factors with load balancing scheme are significantly higher than those without load balancing scheme. For Miami network, the differences in speedup factors are not very large since Miami has a relatively even degree distribution and loads are already fairly balanced without load balancing scheme. However, for real-world skewed

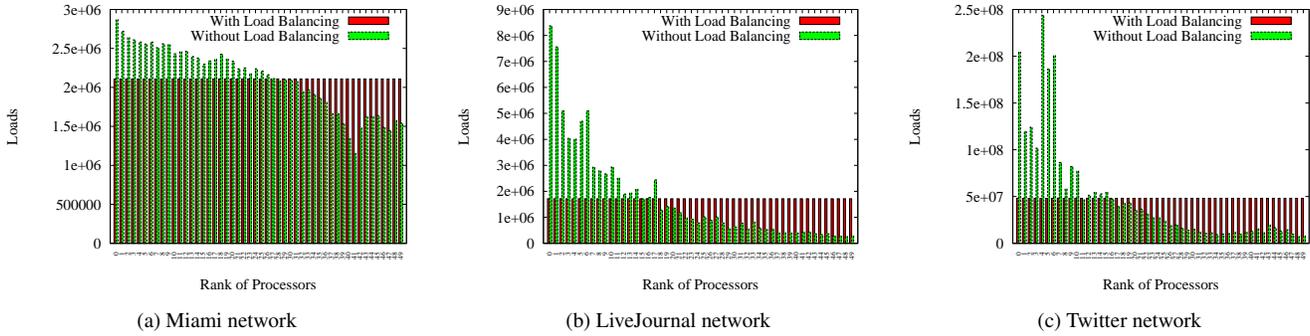


Figure 7: Load distribution among processors for LiveJournal, Miami and Twitter networks by different schemes.

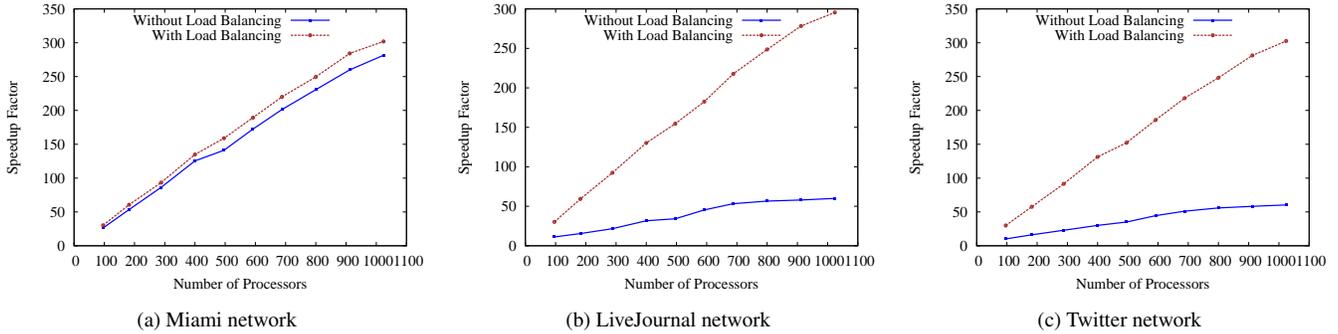


Figure 8: Strong scaling of our algorithm on LiveJournal, Miami and Twitter networks with and without load balancing scheme. Computation of speedup factors includes the cost for load balancing.

networks, our load balancing scheme always improves the speedup quite significantly— for example, with 1024 processors, the algorithm achieves a speedup factor of 297 with load balancing scheme compared to 60 without load balancing scheme for LiveJournal network.

This experiment also demonstrates that our algorithm scales to a large number of processors. The speedup factors continue to grow almost linearly up to 1024 processors.

Comparison between Message-based and External-memory Merging

We compare the runtime and memory usage of our algorithm with both message-based and external-memory merging. Message-based merging is very fast and uses message buffers in main memory for communication. On the other hand, external-memory merging saves main memory by using disk space even though it requires large runtime for I/O operations. Thus these two methods provide desirable alternatives to trade-off between space and runtime. However, as shown in Table 3, message-based merging is significantly faster (up to 20 \times) than external-memory merging albeit taking a little larger space. Thus message-based merging is the preferable method in our fast parallel algorithm.

Network	Memory (MB)		Runtime (s)	
	EXT	MSG	EXT	MSG
Email-Enron	1.8	2.4	3.371	0.078
web-BerkStan	7.6	10.3	10.893	1.578
Miami	26.5	43.34	33.678	6.015
LiveJournal	28.7	42.4	31.075	5.112
Twitter	685.93	1062.7	1800.984	90.894
Gnp(500K, 20)	6.1	9.8	6.946	1.001
PA(5M, 20)	68.2	100.1	35.837	7.132
PA(1B, 20)	9830.5	12896.6	14401.5	1198.30

Table 3: Comparison of external-memory (EXT) and message-based (MSG) merging (using 50 processors).

Weak Scaling

Weak scaling of a parallel algorithm shows the ability of the algorithm to maintain constant computation time when the problem size grows proportionally with the increasing number of processors. As shown in Figure 9, total computation time of our algorithm (including load balancing time) grows slowly with the addition of processors. This is expected since the communication overhead increases with additional processors. However, the growth of runtime of our algorithm is

rather slow and remains almost constant, the weak scaling of the algorithm is very good.

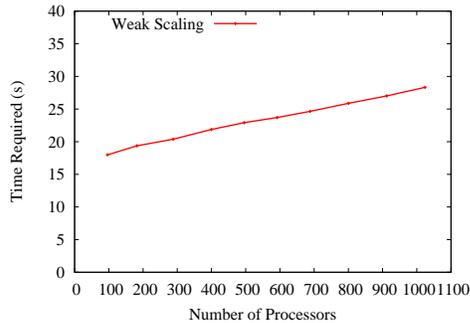


Figure 9: Weak scaling of our parallel algorithm. For this experiment we use networks $PA(x/10 \times 1M, 20)$ for x processors.

CONCLUSION

We present a parallel algorithm for converting *edge-list* of a graph to *adjacency-list*. The algorithm scales well to a large number of processors and works on massive graphs. We devise a load balancing scheme that improves both space efficiency and runtime of the algorithm, even for networks with very skewed degree distributions. To the best of our knowledge, it is the first parallel algorithm to convert edge list to adjacency list for large-scale graphs. It also allows other graph algorithms to work on massive graphs which emerge naturally as edge lists. Furthermore, this work demonstrates how a seemingly trivial problem becomes challenging when we are dealing with Bigdata.

ACKNOWLEDGMENT

We thank our external collaborators, members of Network Dynamics & Simulation Science Laboratory, and anonymous reviewers for their suggestions and comments. This work has been partially supported by DTRA Grant HDTRA1-11-1-0016, DTRA CNIMS Contract HDTRA1-11-D-0016-0001, NSF NetSE Grant CNS-1011769, and NSF SDCI Grant OCI-1032677.

REFERENCES

1. Alam, M., and Khan, M. Efficient algorithms for generating massive random networks. Technical Report 13-064, NDSSL at Virginia Tech, May 2013.
2. Alam, M., Khan, M., and Marathe, M. Distributed-memory parallel algorithms for generating massive scale-free networks using preferential attachment model. In *Proc. of the Intl. Conf. on High Performance Computing, Networking, Storage and Analysis* (2013).
3. Alon, N., Yuster, R., and Zwick, U. Finding and counting given length cycles. *Algorithmica* 17 (1997), 209–223.
4. Aluru, S. Teaching parallel computing through parallel prefix. In *the Intl. Conf. on High Performance Computing, Networking, Storage and Analysis* (2012).
5. Apte, C., et al. Business applications of data mining. *Communications of the ACM* 45, 8 (Aug. 2002), 49–53.
6. Arifuzzaman, S., Khan, M., and Marathe, M. Patric: A parallel algorithm for counting triangles in massive networks. In *Proc. of the 22nd ACM Intl. Conf. on Information and Knowledge Management* (2013).
7. Attewell, P., and Monaghan, D. *Data Mining for the Social Sciences: An Introduction*. University of California Press, 2015.
8. Barabasi, A., et al. Emergence of scaling in random networks. *Science* 286 (1999), 509–512.
9. Barrett, C., et al. Generation and analysis of large synthetic social contact networks. In *Proc. of the 2009 Winter Simulation Conference* (2009), 103–114.
10. Bollobas, B. *Random Graphs*. Cambridge Univ. Press, 2001.
11. Broder, A., et al. Graph structure in the web. *Computer Networks* (2000), 309–320.
12. Chen, J., and Lonardi, S. *Biological Data Mining*, 1st ed. Chapman & Hall/CRC, 2009.
13. Chu, S., and Cheng, J. Triangle listing in massive networks and its applications. In *Proc. of the 17th ACM SIGKDD Conf. on Knowledge Discovery and Data Mining* (2011), 672–680.
14. Chung, F., et al. *Complex Graphs and Networks*. American Mathematical Society, Aug. 2006.
15. Coppersmith, D., and Winograd, S. Matrix multiplication via arithmetic progressions. In *Proc. of the 19th Annual ACM Symposium on Theory of Computing* (1987), 1–6.
16. Girvan, M., and Newman, M. Community structure in social and biological networks. *Proc. Natl. Acad. of Sci. USA* 99, 12 (June 2002), 7821–7826.
17. Kwak, H., et al. What is twitter, a social network or a news media? In *Proc. of the 19th Intl. Conf. on World Wide web* (2010), 591–600.
18. Leskovec, J. Dynamics of large networks. In *Ph.D. Thesis, Pittsburgh, PA, USA*. (2008).
19. Newman, M. Coauthorship networks and patterns of scientific collaboration. *Proc. Natl. Acad. of Sci. USA* 101, 1 (April 2004), 5200–5205.
20. SNAP. Stanford network analysis project. <http://snap.stanford.edu/>, 2012.

A Research Framework for Exascale Simulations of Distributed Virtual World Environments on High Performance Computing (HPC) Clusters

Amit Goel

Institute for Simulation and Training
University of Central Florida
Orlando, FL, USA
amit.goel@ucf.edu

William A. Rivera

Institute for Simulation and Training
University of Central Florida
Orlando, FL, USA
14wrivera@gmail.com

Peter J. Kincaid

Institute for Simulation and Training
University of Central Florida
Orlando, FL, USA
pkincaid@ist.ucf.edu

Waldemar Karwowski

Department of Industrial Engineering and Management Systems
University of Central Florida
Orlando, FL, USA
wkar@mail.ucf.edu

Michele M. Montgomery

Department of Physics
University of Central Florida
Orlando, FL, USA
Michele.M.Montgomery@ucf.edu

Neal Finkelstein

Simulation & Training Technology Center (STTC)
US Army Research Laboratory
Orlando, FL, USA
Neal.Finkelstein@us.army.mil

ABSTRACT

Distributed virtual environment simulators such as OpenSimulator and SecondLife have become popular over the last decade. However, these simulators pose a greater challenge with increased load due to High-Fidelity of simulations, higher number of objects and agents, and increased levels of interactions. In this paper we propose a framework for distributed virtual environment simulator research on high performance computing (HPC) clusters. Our proposed framework would be very useful to virtual world researchers as HPC clusters are more freely available as compared to setting up their own dedicated cluster for virtual environment simulation. Although we implemented and validated our framework on the Stokes Cluster at UCF Advanced Research Computing Center and OpenSimulator virtual world; this framework is generic enough to be extended to other HPC clusters and distributed virtual environment simulators.

Author Keywords

Virtual Worlds, Distributed Virtual Environments, Distributed Simulation, SecondLife, OpenSimulator, High Performance Computing

ACM Classification Keywords

I.6.7 SIMULATION SUPPORT SYSTEMS; I.6.8 TYPES OF SIMULATION: Parallel; I.6.m Miscellaneous

INTRODUCTION

Distributed virtual environment simulators have found an increased usage among education [1], training [7], military¹ [15], medical [3, 7], physics, astronomy[8], arts [4], engineering [4], architecture [4], and many other fields. The challenges associated with these simulators have also proliferated as researchers need to simulate a higher number of objects, agents, and interactions with increased levels of fidelity.

We present a research framework for exascale simulations of distributed virtual world environments on high performance computing (HPC) clusters. In the rest of this section we provide our motivation, research problem statement and related work.

Motivation

Our motivation for using HPC clusters was driven by several factors such as:

- We did not have access to computing machines for building dedicated clusters of distributed virtual environments for experimental simulation purpose.
- Even if we built such dedicated cluster systems their usage would be idle when we are not running any simulations.
- The HPC clusters with huge compute facilities were available, both within our institution and within our institutional network such as at SURAGrid², SSERCA³ and XSEDE⁴.
- Most of HPC clusters used infiniband or other high speed communication networks [9] within cluster. Such high

¹<http://militarymetaverse.org/>

²http://www.suragrid.org/sura_grid.html

³<http://www.sserca.org/>

⁴<https://www.xsede.org/overview>

speed communication network equipment would be much more expensive to setup within a researchers dedicated cluster. Hence using HPC with infiniband or similar high speed/bandwidth networking technologies allows researchers to do exascale experiments with the distributed virtual environment simulations.

Research Problems

During our initial inquiry we came up with the following research problems that we addressed and are reporting in this paper:

- How can we conduct simulation research using OpenSimulator as a distributed virtual environment on the UCF HPC Cluster known as Stokes?
- How do we make this framework generic enough to be available for use on other similar clusters such as BlueWaters⁵ 6?

In this paper we introduce a research framework that addresses these research problems. We have used this framework in our initial experiments and have been incrementally adding advanced features as an ongoing effort to improve future simulation research efforts.

Related Work

We did not find any other similar work for virtual environment simulators being run by researchers on HPC clusters.

We found performance comparisons of virtual world simulators on virtual machines[14]. However, in that work, researchers did not create a research framework that could be reused or generalized for other virtual machine and virtual world software stacks. We propose a generic approach and framework for HPC that can also be customized and used for virtual machines.

We were inspired by work done in Hadoop Map-Reduce Community. Srikrishna et. al. at San Diego Computing Center developed a system called myHadoop[10, 11] that runs on HPC clusters on demand. The myHadoop concept essentially formed the foundation of our framework, as they have similar restrictions. Hadoop or MapReduce environments demand for their own dedicated cluster to be setup, myHadoop system basically runs the Hadoop system on demand on compute nodes allocated by a cluster resource management system and then shuts it down.

Our work is different from myHadoop such that we not only developed a whole set of scripts to configure the OpenSimulator in different modes and run the simulations, but also developed software code to collect the simulator and system statistics. Further, in our configurations we use ethernet network for logging and control connections and high speed infiniband for communicating with the simulation engines, grid services and database.

⁵<http://www.ncsa.illinois.edu/enabling/bluewater>

⁶<https://bluewater.ncsa.illinois.edu/>

Thus we were further motivated to develop this research framework due to lack of similar work for broad usage of virtual worlds research community.

BACKGROUND

Virtual Worlds and Metaverse

For the purpose of our ongoing research and this paper, when we refer to **virtual world**, we mean all sorts of 3D interactive distributed virtual environments such as SecondLife and OpenSimulator.

Furthermore, we use the term Virtual World to represent Multi User Virtual Environments (MUVE), 3D Virtual Environments and Massively Multiplayer Online Game (MMOG), Massively Multiplayer Online Role-Playing Games (MMORPG) and Massively Multiplayer Online Social Games (MMOSG) although we understand there are subtle differences in each of them. For example, MMORPG focus on structured objective-based game play, whereas MMOSG focus on social interaction and communication without a pre-defined structure.

Our definition of virtual worlds draws from several sources of information [16, 2], as cited in this paper:

Definition 1. A **virtual world** is computer-based interactive simulation environment where users take the form of an avatar, own virtual assets, perform virtual actions and interact with other users and computer-controlled actors known as bots or non-player characters (NPC).

Definition 2. **Metaverse** is the collective sum of all virtual worlds, virtual reality and augmented reality on the internet.

The term metaverse was coined in futuristic science fiction novel titled Snow Crash authored by Neal Stephenson in 1992⁷. In the Snow Crash novel human-controlled avatars interact with each other and with computer controlled agents known as bots, in a 3D space that uses the metaphor of the real world.

In virtual worlds users immerse themselves in a 3D environment using avatars to socialize, move around, buy objects, construct objects that they can sell or share, and communicate with other avatars and bots. Thus the immersive and global nature of virtual worlds containing a large number of users and complex interactions with elements of role-playing are attracting researchers from all fields to experiment using virtual worlds [12].

Now let us look at the Virtual World Simulator, known as OpenSimulator, that we used for developing our framework and experiments.

OpenSimulator

OpenSimulator⁸ is a Virtual World Simulator developed by a developer community as an open source project. OpenSimulator builds on top of available components such as LibOpen-

⁷<http://iml.jou.ufl.edu/projects/Fall108/Davis/html/>

⁸<http://www.opensimulator.org>

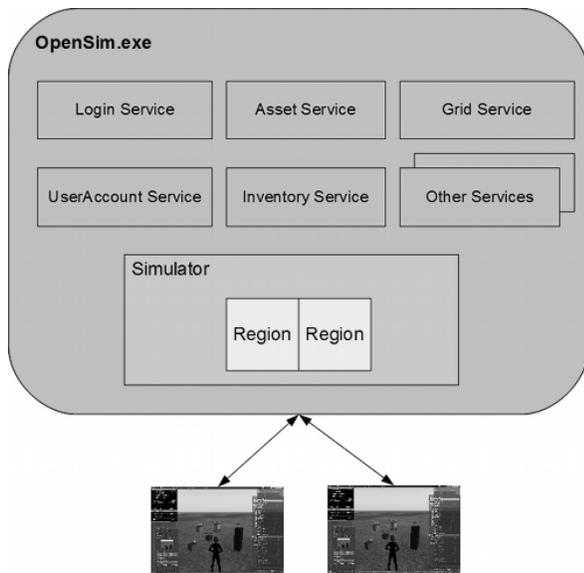


Figure 1: OpenSim Standalone Mode Architecture

Metaverse⁹ and Bullet Physics Engine¹⁰. It can be configured to utilize alternate components that are different from the ones configured by default.

OpenSimulator software stack itself consists of four major components: Database, Simulator Engine, Grid Service Engine and the Viewer. It has built in support for major databases such as MySQL, PostgreSQL and MS SQL, and can be extended to support other database systems. Similarly OpenSimulator based virtual worlds can be accessed using many viewers such as Firestorm and Singularity Viewer. OpenSimulator runs in Standalone or Grid mode.

As shown in Figure 1¹¹, in Standalone mode, OpenSimulator engine (OpenSim.exe) itself provides grid services as well as a simulator and can run multiple regions within one simulator.

Figure 2¹² shows the OpenSimulator architecture in grid mode. In grid mode OpenSimulator engine (OpenSim.exe) only runs one or multiple regions within one simulator. More instances of the OpenSimulator engine can be started on the same or multiple computers to serve more regions. Robust Grid Services are provided as part of the OpenSimulator codebase although other options for Grid Services are available such as SimianGrid¹³.

At the core of OpenSimulator resides a well defined component called regions that has a framework of entities, functions and services that are a basic foundation for input to the scene simulator that can be physics or non-physics based. Region simulators basically talk to other simulators.

⁹<http://openmetaverse.org/projects/libopenmetaverse>

¹⁰<http://bulletphysics.org/wordpress/>

¹¹<http://opensimulator.org/wiki/Configuration>

¹²<http://opensimulator.org/wiki/Configuration>

¹³<https://github.com/openmetaversefoundation/simiangrid>

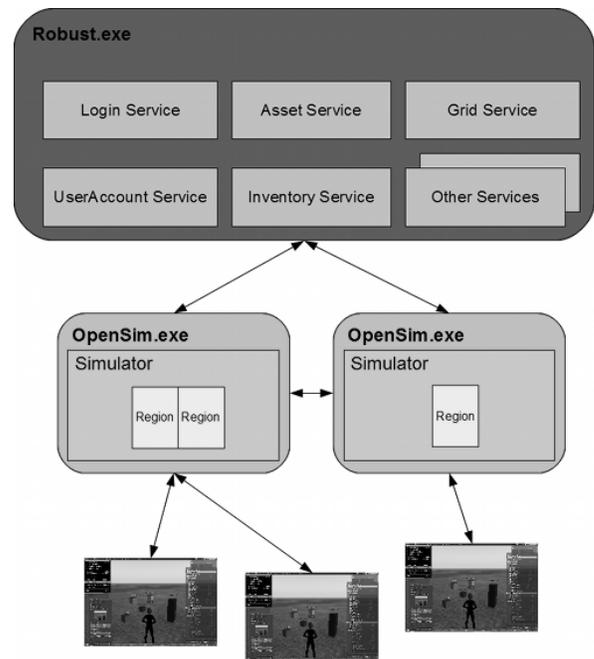


Figure 2: OpenSim Grid Mode Architecture

HPC Clusters

High Performance Computing Clusters such as BlueWaters¹⁴ at University of Illinois National Center for Supercomputing Applications¹⁵, and Stokes¹⁶ at UCF Advanced Research Computing Center, are available to academic researchers almost at no cost but with certain restrictions.

These clusters usually operate in batch mode where a user logs into a head node and submits a job to the Resource Management System on the cluster such as Torque [5], Sun Grid Engine [6] or SLURM [17].

The HPC software stack such as the operating system, resource management system and other components of the stack are made available to be used by all researchers who use the system and thus Virtual World Researchers do not have a choice to change the components of the software stack to fit their needs.

As shown in Figure 3, the UCF Stokes Cluster¹⁷ has four kinds of nodes. Management Nodes run the resource management tools, file system nodes have the shared file storage that is made accessible to all compute nodes, user nodes are used for logging in and submitting jobs by the users, computer nodes are used for running the jobs that required high performance computing resources.

RESEARCH FRAMEWORK FOR HPC CLUSTERS

¹⁴<https://bluewaters.ncsa.illinois.edu/>

¹⁵<http://www.ncsa.illinois.edu/enabling/bluewaters>

¹⁶<http://arcc.ist.ucf.edu>

¹⁷<http://arcc.ist.ucf.edu/wordpress/wp-content/uploads/2012/12/STOKES-Job-Submission.png>

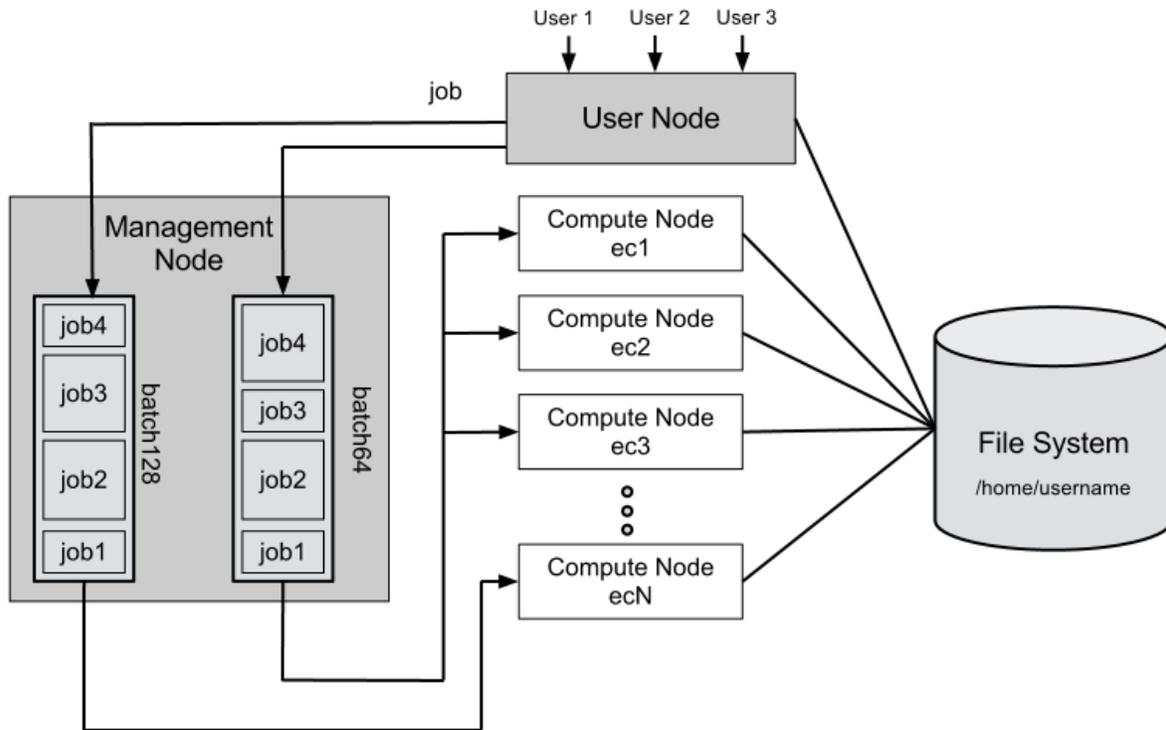


Figure 3: UCF Stokes Cluster Architecture

Essentially components in our research framework can be described in three high level packages:

- Simulation Control and HPC Scripts
- Benchmarks and Configurations
- Monitoring and Logging

Let us elaborate the components in these packages in detail in further subsections. Figure 4 shows these components running on compute node. The dotted lines show ethernet connections between StatServer and Autobot, the solid lines show connection between components on different nodes using high speed infiniband connections.

Simulation Control and HPC Scripts

The simulation of distributed virtual world environments is a bit tricky on HPC clusters. HPC clusters mostly run in a batch mode where the Job runs on compute nodes allocated by resource management system. Not only can you not access the job interactively, but the IP address and node on which the job will execute is unknown in advance. Distributed virtual world environments have their own server configurations. Thus we wrote special HPC scripts using PBS scripts of Torque and shell scripting to implement the Algorithms given in this paper.

The top level control is exerted by Algorithm 1. This algorithm is implemented to be run by the Resource Management

System. It starts the database, opensim engine(s), grid service, StatServers and autobot (described in further subsections). Once the simulation is run for a specified time, this scripts stops all the servers one by one.

The rest of the algorithms are implemented to be executed on their respective nodes. Algorithm 2 is responsible for starting and stopping the database on db-node and Algorithm 4 starts and stops the GridService, if enabled. SimEngine Algorithm 3 not only starts and stops SimEngine and StatServer but also loads the regions into respective SimEngines as per configuration passed by the master control script.

AutoBot is launched on an autobot-node by Algorithm 5. We describe in further subsections the autoboot tool developed as part of this framework.

In the future we plan to extend the framework by allowing for multiple GridServers, AutoBot Servers and file based configuration.

Benchmarks and Configuration

In order to run the simulations in a headless mode, since interactive access to simengine or gridservices are not available, the virtual world needs to be pre-loaded and configured with regions, users, scripts and other objects. Thus as part of developing the framework we selected three scenarios to serve as benchmarks and prepared a configuration with 0-100 users that would be launched by autobot software performing the

Algorithm 1**HPC Virtual World Simulation Control**

```
1: Setup Parameters for Resource Management System
   - Walltime,
   - Nodes // minimum 3 if standalone mode, minimum 4 for grid mode
   - Cores per node
   - Job Name
   - Queue Name
   - Log File Name - e.g. PBS_JOBID.log
2: Count Number of Unique nodes
3: Create nodes-ethernet-ip[] array
4: Create nodes-ib-ip[] array
5: create various environment variables
6: call database-start function on database-node
7: sleep 60 second
8: call grid-start on grid-services-node // if enabled
9: for  $j \leftarrow 1, \text{number of sim} - \text{engine} - \text{nodes}$  do
10:   call opensim-start function on opensim-nodes[j]
11: end for
12: call autobot-start function on autobot-node
13: sleep simulation-time
14: call autobot-stop function on autobot-node
15: for  $j \leftarrow 1, \text{number of sim} - \text{engine} - \text{nodes}$  do
16:   call opensim-stop function on opensim-nodes[j]
17: end for
18: sleep 60 second // wait for simulators to persist info to database
19: grid-stop on grid-services-node // if enabled
20: call database-stop function on database-node
```

Algorithm 2**Database Control Algorithm**

```
1: function DB-START(DB-location, DB-Ethernet-IP, DB-Infiniband-IP)
2:   start database listening on both IP from DB-location
3:   populate OpenSim Database with user and object values
4:   populate other databases such as GridServices Database
5: end function
6: function DB-STOP(DB-location, DB-Ethernet-IP, DB-Infiniband-IP)
7:   drop the databases populated earlier (except log database)
8:   stop database listening on both IP from DB-location
9: end function
```

Algorithm 3**Simulation Engine Control Algorithm**

```
1: function SIM-ENGINE-START(ethernet-ip, ib-ip, sim-engine-config)
2:   configure sim-engine to listen on ib-ip and with sim-engine-config
3:   start SimEngine on ib-ip
4:   start SimEngineStatServer on ethernet-ip
5:   load regions
6: end function
7: function SIM-ENGINE-STOP(ethernet-ip, ib-ip, sim-engine-config)
8:   stop SimEngineStatServer
9:   stop SimEngines
10: end function
```

Algorithm 4Grid Services Control Algorithm

```
1: function GRID-START(ethernet-ip, ib-ip, grid-config)
2:   configure grid-service to listen on ib-ip and with grid-config
3:   start GridService on ib-ip
4:   start GridServiceStatServer on ethernet-ip
5: end function
6: function GRID-STOP
7:   stop GridServiceStatServer
8:   stop GridService
9: end function
```

Algorithm 5AutoBot Control Algorithm

```
1: function AUTOBOT-START(ethernet-ip, ib-ip, autobot)
2:   configure autobot to on ethernet-ip for connecting to StatServer and ib-ip for connecting to regions and grid
3:   start autobot
4: end function
5: function AUTOBOT-STOP
6:   stop autobot
7: end function
```

same or random activities. Further the number of users can be controlled using configuration parameters.

We describe the benchmarks used for the current set of experiments in the experiments and results section. We plan to extend the benchmarks in our proposed framework and augment them with Automated Scenario Generation.

Monitoring and Logging

OpenSimulator provides a number of facilities for collecting data while the simulator is running via http web calls, internal scripts that run using the Linden scripting language (LSL) and remote procedure calls (RPC). Although these mechanisms provide some low level detail they do not provide CPU load information.

In our research we identified the need for tools that can provide operating system information specific to the running simulator on a granular level as well as tools for externally stress testing the system in a programmatic way. To address this issue we built a stress test system that could be configured and deployed withing an HPC system.

For stress testing we developed enhancements to an external client which is now called **AutoBot** that allowed for the ability to create a predefined amount of users. AutoBot allows for the creation of many simulated agents which can be programmed to interact with the environment such as intermittently switching between high physics behaviors. This includes moving sporadically throughout the system and also occasionally interacting with objects by performing simple commands such as grabbing near by objects and jumping.

To collect low level process information a **StatServer** program was created that could be queried dynamically and report process level statistics including memory (shared and virtual) and CPU utilization.

Once deployed the tools allow for the collection of the following information which performs automatically sampling at randomized intervals during simulation:

- Scripted objects count: The amount of scripted objects executed at that given time.
- Resent packets count: The amount of resent packets at that given time.
- Received resent packets count: The amount of received packets from the originating resent ones at that given time.
- Sent pings count: The amount of pings sent at that given time.
- FPS: The frames per second for that the simulator scene at that given time.
- Physics FPS: the physics engines frames per second for that the simulator scene at that given time.
- Agent Updates: The number of agent updates sent at that given time.
- Active scripts count: The number of active scripts running at that given time. This information was constant per scenario complexity as they were loaded with the terrain.
- Agent count: The number of agents in the scene at that given time.
- Agent Speed X: the combined speed of each avatar towards the X axis at that given time.
- Agent Speed Y: The combined speed of each avatar towards the Y axis at that given time.
- Agent Speed Z: The combined speed of each avatar towards the Z axis at that given time.

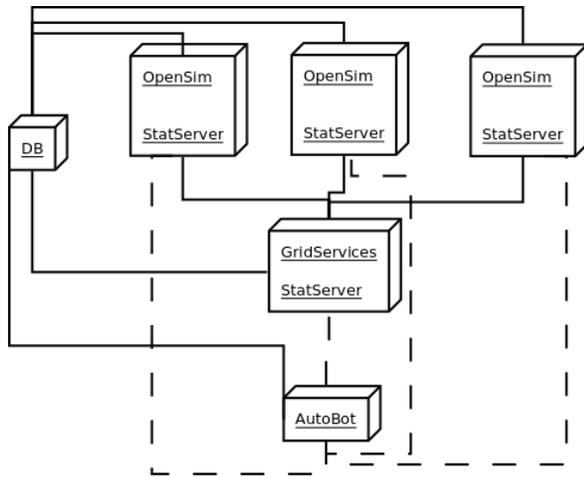


Figure 4: Architecture of Research Framework on HPC Compute Nodes

- Agent Position: The position of each agent in the world at a given time.
- Time: The time intervals were recorded as time stamps down to the second. This was used for informational purposes mostly.
- Memory %: The amount of memory utilization for that process at that given time.
- CPU %: The CPU utilization amount recorded was specific to that process and not all running process on the machine.

EXPERIMENT AND RESULTS

Using the research framework described above we conducted experiments on controllable factors in the scene which included various terrain, running scripts and varying agent density. We used three scenarios of complexity: light, moderate and heavy based on the environment's primitive objects to help understand the role a complex environment impacts simulation performance.

For the heavy scenario a terrain called Business District was used which contained a total of 17,323 primitive objects. This terrain contained different sized buildings that resembled offices, schools and stores. This terrain also included 18 furnished brownstone offices with running scripts for animations. For the moderate scenario a terrain called Boardwalk was used which contained a total of 7,450 primitive objects. This terrain contained piers with the boardwalk stores in the background and a small amount of running scripts. For the light scenario a terrain called Garden Center was used which contained a total of 1,765 primitive objects. This terrain contained a landscape with some trees, a small house and did not have many obstructing areas.

Each of the different complexity scenarios were then run with different sized groups of simulated agents ranging from 20 to 100 in 20 count increments. Before each run the system was updated to its initial state to ensure that each run started with the same initial conditions and remove any bias between runs.

The simulated agents were programmed to move sporadically throughout the system and also interact with the environment by grabbing near by objects and jumping from time to time. Running the 80 and 100 count sized groups resulted in system crashes.

We were thus able to collect and analyze the data from these experiments running on an HPC Cluster using our Research Framework. Given in table 1 is correlation matrix of various fields collected as part of grid partitioning experiments[13]. In using our framework researchers found that as objects enter into an avatars area of activity[13] the amount of updates increases. This was actually not expected as OpenSimulator sends all region updates to each client in the region regardless of avatar proximity. Thus, this research illustrates how our research framework can be used by researchers to run experiments and conduct researchers using HPC resources.

As of now our research framework only collects data, but in the future we plan to integrate it with Hadoop, R or other big-data processing software to generate statistical inferences and analysis automatically at the end of each simulation run.

CONCLUSIONS AND FUTURE WORK

We contributed to high performance computing, virtual worlds, distributed virtual environments and distributed simulations research by providing a generic research framework for HPC clusters. Using our framework, OpenSimulator and other distributed virtual environment simulators can scale the simulations to exascale levels using the raw power of HPC.

Our ongoing research in scalability of virtual world simulators and distributed virtual environments continues to look at other challenges for scaling such research to an exascale level.

We plan to improve our framework by applying for resource grants at Bluewaters and Xsede resources. Further we also plan to generalize the framework for use across HPC, virtual machines and dedicated cluster environments.

We are also looking at standardizing the benchmarks, configurations and scenarios for various kinds of usage scenarios such as games, social interactions, military combat, cultural training and medical simulations. We plan to use the automated scenario generation from configuration files for developing our configurable flexible benchmark suite.

ACKNOWLEDGMENTS

The authors acknowledge Mr. Doug Maxwell and Dr. Neal Finkelstein of Simulation and Training Technology Center (STTC) of United States Army Research Laboratory¹⁸ for generously providing computational resources and support that have contributed to results reported herein.

The authors would also like to acknowledge the University of Central Florida Advanced Research Computing Center for providing computational resources on Stokes High Performance Computing Cluster¹⁹.

¹⁸<http://www.arl.army.mil>

¹⁹<http://arcc.ist.ucf.edu>

	Sent Pings	FPS	Physics FPS	Agent Updates	Objects	Speed	AOA	Agents	Active Scripts	CPU	Memory
Sent Pings	1.000	0.037	0.034	-0.059	-0.203	0.091	-0.116	0.036	-0.222	0.022	-0.196
FPS	0.037	1.000	0.992	-0.373	-0.360	-0.273	-0.525	-0.371	-0.358	-0.417	-0.455
Physics FPS	0.034	0.992	1.000	-0.348	-0.352	-0.256	-0.501	-0.347	-0.350	-0.409	-0.441
Agent Updates	-0.059	-0.373	-0.348	1.000	0.050	0.751	0.721	0.945	0.059	0.665	0.299
Objects	-0.203	-0.360	-0.352	0.050	1.000	-0.162	0.627	-0.064	0.996	0.024	0.961
Speed	0.091	-0.273	-0.256	0.751	-0.162	1.000	0.319	0.900	-0.205	0.590	0.009
AOA	-0.116	-0.525	-0.501	0.721	0.627	0.319	1.000	0.585	0.651	0.481	0.811
Agents	0.036	-0.371	-0.347	0.945	-0.064	0.900	0.584	1.000	-0.079	0.687	0.169
Active Scripts	-0.222	-0.358	-0.350	0.059	0.996	-0.205	0.651	-0.079	1.000	0.018	0.964
CPU	0.022	-0.417	-0.409	0.665	0.024	0.590	0.481	0.687	0.018	1.000	0.193
Memory	-0.196	-0.455	-0.441	0.299	0.961	0.009	0.811	0.169	0.964	0.193	1.000

Table 1: Correlation Matrix

REFERENCES

- Balula, A., and Moreira, A. e-Teaching evaluation in higher education. In *Evaluation of Online Higher Education: Learning, Interaction and Technology*, SpringerBriefs in Education. Springer International Publishing, 2014, 1–11.
- Bell, M. W. Toward a Definition of "Virtual Worlds". *Journal of Virtual Worlds Research* 1, 1 (2008).
- Diener, S., Windsor, J., and Bodily, D. Design and development of clinical simulations in SecondLife. In *Proceedings of the EDUCAUSE Australasia Conference* (2009), 1–13.
- Duran, J., and Villagrasa, S. Teaching 3D arts using game engines for engineering and architecture. In *Proceedings of the 5th International Conference on Virtual, Augmented and Mixed Reality (VAMR 2013) held as part of HCI International (HCII 2013)*, Springer (2013), 113–121.
- Emenecker, W., Jackson, D., Butikofer, J., and Stanzione, D. Dynamic Virtual Clustering with Xen and Moab. In *Frontiers of High Performance Computing and Networking ISPA 2006 Workshops*, G. Min, B. D. Martino, L. T. Yang, M. Guo, and G. Rünger, Eds. Springer, 2006, 440–451.
- Gentzsch, W. Sun Grid Engine: towards creating a compute power grid. *Proceedings of the 1st IEEE/ACM International Symposium on Cluster Computing and the Grid* (2001), 35–36.
- Gomes, T., Abade, T., Campos, J. C., Harris, M. D., and Silva, J. L. A Virtual Environment based Serious Game to Support Health Education. *ICST Transactions on Ambient Systems* 1, 3 (Mar. 2014), e5.
- Henckel, A., and Lopes, C. V. StellarSim : A plug-in architecture for scientific visualizations in virtual worlds. In *Proceedings of the Conference on Facets of Virtual Environments (FaVE 2009)*, Springer (2010), 106–120.
- Kerbyson, D. J., Barker, K. J., Vishnu, A., and Hoisie, A. A performance comparison of current HPC systems: Blue Gene/Q, Cray XE6 and InfiniBand systems. *Future Generation Computer Systems* 30 (Jan. 2014), 291–304.
- Krishnan, S. myHadoop 0.2a : Hadoop-on-demand on traditional HPC resources. Tech. rep., San Diego Supercomputing Center, 2012.
- Krishnan, S., Tatineni, M., and Baru, C. myHadoop - Hadoop-on-Demand on Traditional HPC Resources. Tech. rep., San Diego Supercomputing Center, 2011.
- McKee, H. A., and Porter, J. E. Playing a Good Game : Ethical Issues in Researching MMOGs and Virtual Worlds. *International Journal of Internet Research Ethics* 2, 1 (2009).
- Rivera, W. A., and Goel, A. Calculating Grid Partitioning Costs of Distributed Virtual World Simulation Systems. In *Submitted to SpringSim 2015* (2015).
- Sanatinia, A., Oliver, I. A., Miller, A. H. D., and Allison, C. Virtual machines for virtual worlds. In *Proceedings of the 2nd International Conference on Cloud Computing and Services Science (CLOSER 2012)*, F. Leyman, I. Ivanov, M. van Sinderen, and T. Shan, Eds., SciTePress (2012), 108–113.
- Scacchi, W., Brown, C., and Nies, K. Exploring the Potential of Virtual Worlds for Decentralized Command and Control Point of Contact. In *Proceedings of the 17th International Command and Control Research and Technology Symposium (ICCRTS 2012)*, DTIC Document (2012).
- Schroeder, R. Defining virtual worlds and virtual environments. *Journal of Virtual Worlds Research* 1, 1 (2008).
- Yoo, A. B., Jette, M. A., and Grondona, M. SLURM : Simple Linux Utility for Resource Management. In *Job Scheduling Strategies for Parallel Processing*, D. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds., Lecture Notes in Computer Science. Springer, 2003, 44–60.

Fast Sparse Matrix Multiplication on GPU

Lukas Polok

Brno University of
Technology, Faculty of
Information Technology
Bozetechova 2, 612 66 Brno,
Czech Republic
ipolok@fit.vutbr.cz

Viorela Ila

Brno University of
Technology, Faculty of
Information Technology
Bozetechova 2, 612 66 Brno,
Czech Republic
ila@fit.vutbr.cz

Pavel Smrz

Brno University of
Technology, Faculty of
Information Technology
Bozetechova 2, 612 66 Brno,
Czech Republic
smrz@fit.vutbr.cz

ABSTRACT

Sparse matrix multiplication is an important algorithm in a wide variety of problems, including graph algorithms, simulations and linear solving to name a few. Yet, there are but a few works related to acceleration of sparse matrix multiplication on a GPU. We present a fast, novel algorithm for sparse matrix multiplication, outperforming the previous algorithm on GPU up to $3\times$ and CPU up to $30\times$. The principal improvements include more efficient load balancing strategy, and a faster sorting algorithm. The main contribution is design and implementation of efficient sparse matrix multiplication algorithm and extending it to sparse block matrices, which is to our best knowledge the first implementation of this kind.

Author Keywords

parallel sparse matrix multiplication; parallel linear algebra; matrix-matrix multiplication; GPGPU

ACM Classification Keywords

G.1.3 Numerical Linear Algebra: Sparse, structured, and very large systems (direct and iterative methods); G.4 Mathematical Software: Parallel and vector implementations

INTRODUCTION

This paper presents a novel and highly efficient parallel algorithm for sparse matrix multiplication. Sparse matrix-matrix multiplication is an important algorithm, useful in a wide variety of scientific tasks, including among others computational chemistry and physics, graph contraction, breadth-first search from multiple vertices, algebraic multigrid methods, finite element methods or solving (non)linear systems using Schur complement [29].

The sparse matrix algorithms are usually tightly coupled to the sparse matrix storage formats they use. Two of the popular formats are compressed sparse column (CSC) [12] and compressed sparse row (CSR). Those are closely related; matrices stored in one are transposes of the matrices stored in the other. CSC stores matrices as a vector of prefix sums of numbers of nonzero elements in each column and two vectors storing element values and their respective rows. It is common for the elements in each column to be ordered by their row number. The use of the CSC format is assumed in the rest of this paper, unless specified otherwise.

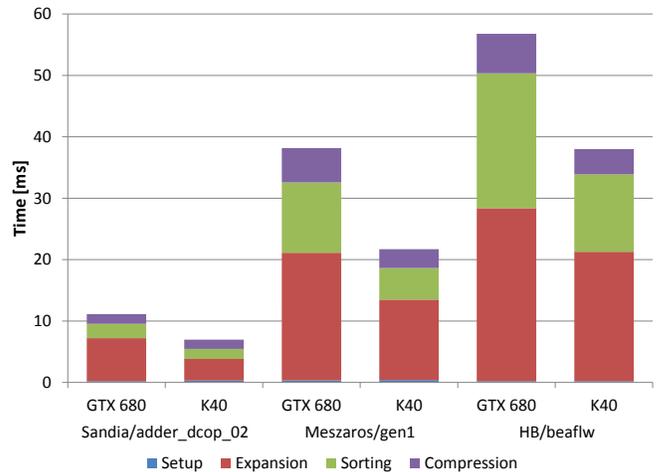


Figure 1. Time of different stages of the proposed algorithm.

Let us recall that in matrix multiplication $C = A \cdot B$, each element of the product $C_{i,j}$ is a sum of products of the corresponding elements in the i^{th} row of A and the j^{th} column of B . The number of columns of A must match the number of rows of B . In CSC, it is straightforward to look up elements by column ($O(1)$) but not to look up elements by row ($O(N)$ in the number of nonzero elements), which would be needed to calculate the elements of C in ordered fashion (*gather*).

The original algorithm for sequential sparse matrix multiplication [16] is implemented e.g. in the popular CSparse package [12] (used by Google's Ceres solver and Street View), and is work-efficient in terms of its complexity being proportional to the number of floating point operations (FLOP). It is worth mentioning that this level of efficiency is only reached for the price of calculating a partially unordered representation of the product, which is still useful in practice, but it is not the canonical form.

The algorithm [16] is efficient by traversing the elements of B column by column (assuming the CSC storage is used; for CSR all the terms are transposed), where each element $B_{i,j}$ multiplies all the elements of A in the i^{th} column (the one corresponding to the *row* of the particular element $B_{i,j}$). Many of the other sparse matrix multiplication algorithms use this strategy. It produces partially ordered partial products (*scatter*), which need to be summed up. The authors of [16]

came up with an elegant way of quickly merging these partially ordered sequences to form the (unordered) result.

Parallel sparse matrix multiplication algorithms (PSPGEMM in BLAS terminology), however, generally decompose the matrices to band or block submatrices and distribute the computation of the partial products to different processors. Similarly like in the previous case, the results need to be merged to form the final product, using sparse matrix addition in this case. This approach is further referred as a coarse-grain work subdivision, since the submatrices are typically relatively large. Packages [4, 2] use this approach.

RELATED WORK

Unlike dense matrix multiplication [3, 13, 9, 15, 14] which is very well researched and widely understood, sparse matrix multiplication [8, 7, 25] is much more challenging - and even more so in the hardware. Many papers titled "sparse matrix multiplication" actually refer to sparse matrix-dense matrix multiplication [17, 28], which is an extension of sparse matrix-vector multiplication (PSPGEMV or PCSRMM), an equally useful but nonetheless different algorithm.

The work of Buluç [8] discusses the challenges of designing and implementing scalable sparse matrix multiplication in distributed memory systems. Coarse-grain 1D decomposition of the work is considered, and two novel 2D algorithms are presented. The identified challenges are the load imbalance, the amount of work for partial result reduction and the communication overheads.

Matam et. al. [17] explore several variations of the work division in a hybrid CPU + GPU algorithm: row-column, column-row and row-row. Therefore, the technique is based on the coarse grain algorithm. A heuristic is proposed for the fastest row-row case that efficiently balances computational load between the CPU and the GPU. The load balancing is further extended for a special case of banded matrices. The work contains highly efficient implementations of both multiplication of two sparse matrices and a sparse with a dense matrix. The sparse kernel achieves 240.663 MFLOP/s on average on matrices from the SNAP dataset, using quad-core Intel Core i7 920 CPU and Tesla C2050 GPU.

The work of Bell [5] is strongly influential in the context of the later developments of GPU algorithms. It proposes the Expansion Sorting Compression (ESC) algorithm. The expansion stage is based on the scattering of partial products to a matrix stored in the triplet form (also known as the COO format), using the same operation ordering as used in [16]. To convert to CSC, the partial products need to be sorted and the entries contributing to the same element of the product need to be *reduced* (summed up) at sorting and compression stages. The parallel primitives considered here are amenable to fine-grain work distribution. The proposed method is also inspired by the ESC algorithm.

The work of Bell was further refined in [10]. Their implementation is public as [1] and it was used for comparisons with the algorithm proposed in this paper. It focuses more intensively on the GPU platform-specific optimizations, such as

avoiding passing data through global memory in favor of local memory and registers, especially in the sorting stage. The CSR storage is used, which is reflected in the three following paragraphs.

A permutation matrix is introduced, which orders the left operand by the work required to process a single row, facilitating load balancing. The product is later reordered, but the cost of doing so is reportedly relatively low.

The memory traffic of the expansion phase is further optimized for more regular coalesced accesses by casting the expansion process as a depth-first search on a layered bipartite graphs of the nonzero elements of both factors.

The sorting phase is also optimized, by realizing that the produced expansion is partially sorted, the expansions of individual rows are contiguous, and only intra-row sorting is required. This is implemented as sorting many rows in the local memory at once. Very long rows which would not fit in the local memory are sorted using a global sort. Further reduction in sorting is achieved by a priori knowledge of the distribution of the bits of the sorted keys, and by copying lower bits of column indices to unused upper bits of row indices, effectively avoiding to have to sort simultaneously or sequentially by two keys.

Well known parallel algorithms, such as parallel sum (*reduction*) and parallel prefix sum (*scan*) [6, 27] or parallel sort [26, 18, 19] are used by the proposed technique. In the remainder of this paper, it is assumed that the reader is familiar with them.

In our previous work, we showed the usefulness of sparse *block* matrices in solving nonlinear least squares problems [23, 22] such as Simultaneous Localization and Mapping (SLAM) in robotics or Bundle Adjustment (BA) and Structure from Motion (SfM) in computer vision, where the block structure naturally occurs. Some instances of Finite Element Method (FEM) problems may also exhibit block structure. We also showed performance gains of performing arithmetic operations exploiting the block structure [21] on CPU. In this paper, we propose an efficient algorithm for parallel sparse matrix multiplication which also extends to sparse block matrices, which is to our best knowledge the first implementation of *blockwise* PSPGEMM in hardware (note that sparse block matrix-vector multiplication was recently implemented on GPU [24]). Our implementation, unlike some others, runs entirely in the GPU, leaving the CPU available for other tasks.

The remainder of the paper is structured as follows. The following section contains the analysis of the algorithm and possible improvements. Section Implementation details the proposed implementation and optimizations used. Section Results shows the performance of the proposed solution through benchmarks and time comparisons with the existing implementations. Conclusions and future work are given in Section Conclusions and Future Work

ALGORITHM DESIGN

The algorithm introduced in this paper is based on the ESC algorithm [5, 10]. However, the focus is on removing load

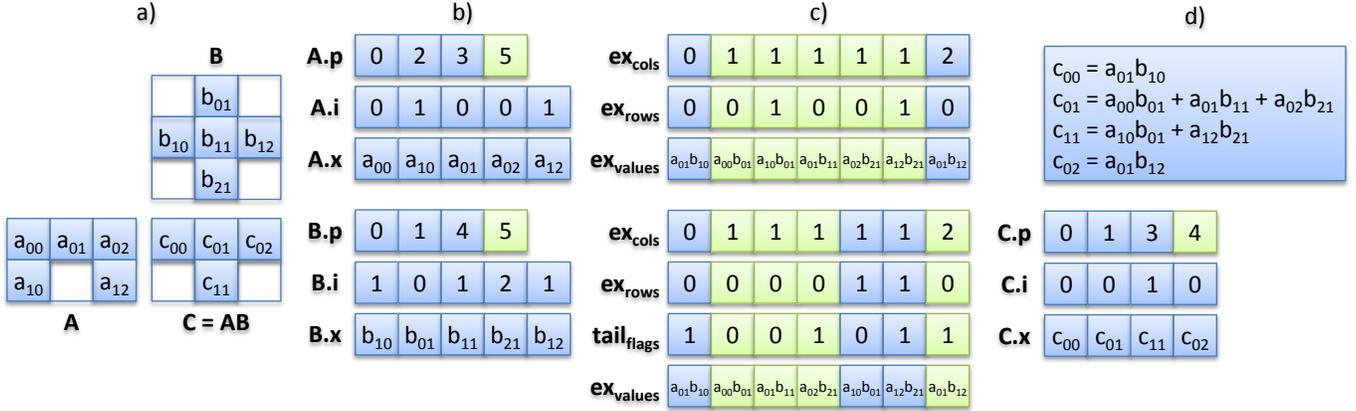


Figure 2. Data at the individual stages of the ESC algorithm²: a) the factors and their product, b) CSC representation of the factors, c) top: expansion of the product, segments of product columns indicated by alternating color, bottom: sorted expansion, segments of product elements indicated by alternating color, d) values of the product and its final CSC form.

imbalances and on simplicity, as especially the improved ESC algorithm in [10] handles many special cases, depending on the memory space (local or global) and granularity (thread, warp or thread group) of each particular operation. In contrast, the proposed implementation only requires six custom kernels, some of which are merely a fusion of multiple general purpose operations such as scan, created for performance purposes only.

Expansion Stage

Although the first conceptual stage of the algorithm is expansion, on GPU it is not possible to directly proceed, without first knowing its size, as all the memory needs to be allocated before starting the computation. From [16], it is trivial to derive the exact size of the expansion:

$$expansion(\mathbf{A}, \mathbf{B}) = \sum_{j=1}^{cols(\mathbf{B})} \sum_{k=1}^{nnzc(\mathbf{B}, j)} nnzc(\mathbf{A}, row(\mathbf{B}, j, k)), \quad (1)$$

where $cols(\cdot)$ gets the number of columns of a matrix, $nnzc(\cdot, \cdot)$ returns the number of nonzero elements in a specified column of a specified matrix, and $row(\cdot, \cdot, \cdot)$ is the row of the given element in a column of a matrix. Note that all those are $O(1)$ array look-ups if the matrix is stored in CSC format. Also note that the expansion size is closely related to the number of FLOPs required to carry the multiplication out.

The expansion size dictates the memory cost of the ESC algorithm (the proposed variant as well as [5, 10]). Fig. 3 plots a ratio of expansion size to the number of nonzeros in the product. In certain cases $100\times$ more storage than the final product is required (please, refer to Section Results for the description of the dataset). Fortunately, it is possible to transparently subdivide the product by cutting the \mathbf{B} matrix to several column slices, producing one slice of the product at a time.

The choice of granularity of expansion is crucial to load balancing. The proposed algorithm achieves perfect load balancing in the expansion stage by using granularity of individual

²An interactive demonstrator is available online at <http://www.fit.vutbr.cz/~ipolok/esc>.

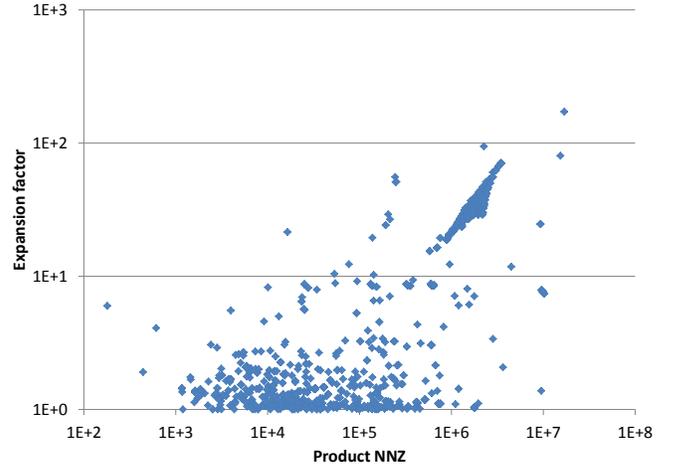


Figure 3. Expansion factor by the number of product NNZ.

scalar products. To do that, it is necessary for each thread to find the elements of \mathbf{A} and \mathbf{B} to process. Here, the *interpolation search* [20] algorithm is employed. It is a special case of binary search where the pivot is chosen based on linear interpolation of the values of the endpoints of the searched interval with the needle as the argument. The average complexity is $O(\log \log N)$, worst case being $O(N)$ (for comparison, binary search is $O(\log N)$ in both cases). Interpolation search is not popular on CPU, as the linear interpolation is too expensive to outperform a regular binary search. However, it is a perfect fit for GPU where linear interpolation nicely hides under memory access latency and allows to find the needle in fewer steps, with much less branching.

The expanded scalar products are essentially the product matrix in the COO format; they can be stored in three vectors of the same length, ex_cols contains columns of the elements, ex_rows contains rows of the elements and ex_values contains values of the elements (see Fig. 2c). Note that the sparse multiplication algorithm generates a partially ordered expansion, where ex_cols is ordered and ex_rows consists of many short ordered runs (given by the rows of elements in the columns of \mathbf{A} , which are typically ordered).

Sorting Stage

The approach in [5] is to use a single global sort. On GPU, the most efficient sort implementations use radix sort [18] with complexity $O(kN)$ where k is proportional to the number of bits of the key. In the case of keys generated by the expansion, the number of bits is given by base 2 logarithm of the number of rows and columns, respectively, and this knowledge can be used to accelerate the sort.

The radix sort may, however, not be the most efficient for a sequence which is already nearly sorted. As the sort starts, the expansion will be first ordered by the least significant bits of the keys, corresponding to the row indices. This will shuffle the column indices which were already ordered at the beginning. The elements are moved by long distances, leading to large amount of potentially uncoalesced global memory traffic. These will be ordered again in the later stages of the sort, by the most significant bits of the keys which correspond to the column indices, leading to long distance movement again.

The radix sort is efficient on GPU if the sorted elements are only reordered by small distances, as such reordering can be performed in the local memory. This is achieved by sorting it in segments corresponding to the individual columns of the product (Fig. 2c top), instead of sorting the whole expansion at once. The individual segments can be sorted in parallel. Now the elements are only reordered by relatively short distances, leading to better write coalescing and leaving ample opportunity to do the sorting in the local memory. However, load balancing issues arise, as the lengths of expansions of the individual columns can vary wildly [10].

In the context of GPU computing, some operations have *segmented* variants, e.g. a *segmented scan*. Its input is a vector of values to calculate the scan of, and a vector of *head flags*, a binary vector with ones at the positions of segment starts. Note that segmented operation is performed on the bulk of data rather than on the individual segments, and thus requires no explicit load balancing. Unfortunately, radix sort is not a good candidate for segmented implementation, as it would lead to both runtime and space tolls: the key bit histograms would need to be evaluated per each segment and the reordering would also need to take place per segment, leading to more load balancing issues. Fortunately, for merge sort, segmented variants exist⁰, and the performance toll, compared to the non-segmented variant, is negligible. By using segmented sort, the time of the sorting stage was significantly reduced; one can compare Fig. 1 where sorting takes only 34%, with Fig. 4 in [10] where it is closer to 63% of the total runtime.

Compression Stage

Once the expansion is sorted, the compression is a simple task of calculating sums of elements with the same row and column, which are now in contiguous segments of the expansion (see Fig. 2c bottom). A simple segmented reduction can be used to calculate the sums, while the head flags can be calculated as a difference of row and column numbers between

⁰One such implementation can be found at http://nvlabs.github.io/moderngpu/seg_sort.html.

consecutive expansion elements. Note that similarly to expansion stage, the size of the compressed form needs to be calculated first (e.g. as a sum (reduction) of the head flags) so that the memory to store the results can be allocated, unless the size of the product is known beforehand.

When handling matrices with large elements, such as long double or especially block matrix elements (i.e. dense blocks), it is beneficial to reorder the operations slightly: instead of storing the *values* of the partial products in the expansion, store only the *pointers* to the operands and calculate the products themselves during compression. This reduces both the size of the expansion and the memory traffic of sorting it.

IMPLEMENTATION

The proposed algorithm was implemented in OpenCL, and is presented in Algorithms 1, 2 and 3. This separation to parts is given by the need to allocate memory, which requires CPU intervention. The algorithm therefore requires GPU - CPU synchronization twice, at the beginning and after the end of Algorithm 2. This may be omitted if the allocation sizes or their upper bounds are known beforehand. Note that all the allocated buffers reside in the GPU memory.

In the algorithms, several conventions are followed. For any matrix \mathbf{M} stored in the CSC format, $\mathbf{M.p}$ is the prefix sum of nonzeros in each of its columns, $\mathbf{M.i}$ is the vector of row indices of nonzero elements and $\mathbf{M.x}$ is the corresponding vector of values of the elements. The parallel GPU *kernel* calls are denoted by **kernel**, and the (one-dimensional) execution domain is specified as $i = 0 \dots N$, where i is the name of the variable holding the thread id, and N is the required number of threads (thread with id $N - 1$ is the last thread).

In the setup stage (Algorithm 1), the \mathbf{b}_{cols} vector is filled with column indices of each corresponding element of \mathbf{B} , making \mathbf{B} available in both COO (intermediate) and CSC (input) formats. This allows $O(1)$ lookup of column of any element of \mathbf{B} in the later stages of the algorithm. Additionally, each element of $\mathbf{b}_{\text{prods}}$ contains the amount of work required to multiply all the *preceding* elements of \mathbf{B} . This will be further used to facilitate load balancing at the expansion stage. The last element contains the total amount of work, which equals

Algorithm 1 Setup stage of PSpGEMM.

```
1: function GEMM(A, B)
2:    $\mathbf{b}_{\text{cols}} = \text{ALLOCINT}(\text{NNZ}(\mathbf{B}))$ 
3:    $\mathbf{b}_{\text{prods}} = \text{ALLOCINT}(\text{NNZ}(\mathbf{B}) + 1)$ 
4:   kernel ( $i = 0 \dots \text{NNZ}(\mathbf{B})$ )
5:      $\mathbf{b}_{\text{cols}}[i] = 0$ 
6:      $\text{row} = \mathbf{B.i}[i]$ 
7:      $\mathbf{b}_{\text{prods}}[i] = \mathbf{A.p}[\text{row} + 1] - \mathbf{A.p}[\text{row}]$ 
8:   end kernel  $\triangleright$  the last element of  $\mathbf{b}_{\text{prods}}$  not initialized
9:   kernel ( $i = 0 \dots \text{COLS}(\mathbf{B})$ )
10:     $\mathbf{b}_{\text{cols}}[\mathbf{B.p}[i + 1] - 1] = 1$ 
11:  end kernel
12:   $\mathbf{b}_{\text{cols}} = \text{EXCLUSIVE SCAN}(\mathbf{b}_{\text{cols}})$ 
13:   $\mathbf{b}_{\text{prods}} = \text{EXCLUSIVE SCAN}(\mathbf{b}_{\text{prods}})$ 
14:   $\text{exp\_size} = \mathbf{b}_{\text{prods}}[\text{NNZ}(\mathbf{B})] \triangleright$  expansion size
```

the expansion size. Note that the kernel at line 9 needs to be modified if \mathbf{B} is known to be rank deficient (then the number of succeeding empty columns needs to be added to each 1 in $\mathbf{b}_{\text{prods}}$, and care must be taken to not write to index -1). These changes were omitted in sake of space.

The expansion stage (Algorithm 2) begins by allocation of the arrays to hold the expanded values. The expansion is performed by the number of threads necessary to saturate the GPU (denoted $GPU_{\text{hardware threads}}$), or less if the expansion is smaller than that. Each thread will calculate the same number of scalar products, as discussed in Section Expansion Stage A range of scalar products to carry out (*begin*, *count*) is allocated for each thread, which then looks up $\mathbf{b}_{\text{prods}}$ for the element of \mathbf{B} where to start multiplying (line 22). UPPER_BOUND is a standard binary search function: for an ordered vector and a value, it returns the right-most position where this value could be inserted without violating the or-

Algorithm 2 Expansion and sorting stages.

```

15:   $\mathbf{ex}_{\text{cols}} = \text{ALLOCINT}(exp\_size)$ 
16:   $\mathbf{ex}_{\text{rows}} = \text{ALLOCINT}(exp\_size)$ 
17:   $\mathbf{ex}_{\text{values}} = \text{ALLOCFLOAT}(exp\_size)$ 
18:   $\mathbf{ex}_{\text{hf}} = \text{ALLOCBIT}(exp\_size)$   $\triangleright$  head flags bit array
19:  kernel ( $i = 0 \dots (N = GPU_{\text{hardware threads}})$ )
20:     $begin = \lfloor exp\_size \cdot i / N \rfloor$ 
21:     $count = \lfloor exp\_size \cdot (i + 1) / N \rfloor - begin$ 
22:     $elemB = \text{UPPER\_BOUND}(\mathbf{b}_{\text{prods}}, begin) - 1$ 
23:     $col\_skip = begin - \mathbf{b}_{\text{prods}}[elemB]$ 
24:    for ( $prod = 0; prod < count; ++ elemB$ ) do
25:       $rowB = \mathbf{B}.i[elemB]$ 
26:       $elemA = col\_skip + \mathbf{A}.p[rowB]$ 
27:       $endA = \mathbf{A}.p[rowB + 1]$ 
28:      while ( $elemA < endA$  and  $p < count$ ) do
29:         $dest = begin + p$ 
30:         $cur\_col = \mathbf{ex}_{\text{cols}}[dest] = \mathbf{b}_{\text{cols}}[elemB]$ 
31:         $\mathbf{ex}_{\text{rows}}[dest] = \mathbf{A}.i[elemA]$ 
32:         $\mathbf{ex}_{\text{values}}[dest] = \mathbf{A}.x[elemA] \cdot \mathbf{B}.x[elemB]$ 
33:         $\mathbf{ex}_{\text{hf}}[dest] = cur\_col > \mathbf{b}_{\text{cols}}[elemB - 1]$ 
34:         $++ elemA, ++ prod$ 
35:      end while
36:       $col\_skip = 0$   $\triangleright$  skip in the first iteration only
37:    end for
38:  end kernel
39:   $\text{SEGMENTEDSORT}(\mathbf{ex}_{\text{hf}}, \mathbf{ex}_{\text{rows}}, \mathbf{ex}_{\text{values}})$ 
40:   $tail\_blocks = \lceil exp\_size / block\_size \rceil$ 
41:   $\mathbf{tail\_counts} = \text{ALLOCINT}(tail\_blocks + 1)$ 
42:     $\triangleright$  or reuse  $\mathbf{b}_{\text{prods}}$  which is not needed below
43:  kernel ( $i = 0 \dots exp\_size - 1$ )
44:    local int flags[ $block\_size$ ]  $\triangleright$  in local memory
45:     $\mathbf{flags}[i] = \mathbf{ex}_{\text{cols}}[i] < \mathbf{ex}_{\text{cols}}[i + 1]$  or
46:       $\mathbf{ex}_{\text{rows}}[i] < \mathbf{ex}_{\text{rows}}[i + 1]$ 
47:     $g = \lfloor i / block\_size \rfloor$   $\triangleright$  cooperating thread group
48:     $\mathbf{tail\_counts}[g] = \text{COOPERATIVE\_REDUCE}(\mathbf{flags})$ 
49:  end kernel
50:   $\mathbf{tail\_counts} = \text{EXCLUSIVESCAN}(\mathbf{tail\_counts})$ 
51:   $product\_NNZ = \mathbf{tail\_counts}[tail\_blocks] + 1$ 

```

Algorithm 3 Compression stage.

```

50:   $\mathbf{C}.p = \text{ALLOCINT}(\text{COLS}(\mathbf{B}) + 1)$ 
51:   $\mathbf{C}.i = \text{ALLOCINT}(product\_NNZ)$ 
52:   $\mathbf{C}.x = \text{ALLOCFLOAT}(product\_NNZ)$ 
53:  kernel ( $i = 0 \dots exp\_size - 1$ )
54:     $g = \lfloor i / block\_size \rfloor$   $\triangleright$  cooperating thread group
55:     $col\_tail = \mathbf{ex}_{\text{cols}}[i] < \mathbf{ex}_{\text{cols}}[i + 1]$ 
56:     $elem\_tail = \mathbf{ex}_{\text{rows}}[i] < \mathbf{ex}_{\text{rows}}[i + 1]$  or  $col\_tail$ 
57:    local int flags[ $block\_size$ ]  $\triangleright$  in local memory
58:     $\mathbf{flags}[i] = elem\_tail$ 
59:     $\mathbf{flags} = \text{COOPERATIVE\_SCAN}(\mathbf{flags})$ 
60:     $compressed\_index = \mathbf{tail\_counts}[g] + \mathbf{flags}[i]$ 
61:    if ( $elem\_tail$  and  $i < exp\_size$ ) then
62:       $\mathbf{C}.i[compressed\_index] = i$   $\triangleright$  write indices of
63:    end if  $\triangleright$  reduced values of elements in expansion
64:    if ( $col\_tail$  and  $i < exp\_size - 1$ ) then
65:       $\mathbf{C}.p[\mathbf{ex}_{\text{cols}}[i] + 1] = compressed\_index + 1$ 
66:    end if  $\triangleright$  write positions of beginnings of columns
67:  end kernel
68:   $\mathbf{C}.p[0] = 0$   $\triangleright$  need to write this explicitly
69:   $\mathbf{ex}_{\text{values}} = \text{SEGMENTEDREDUCTION}(\mathbf{C}.i, \mathbf{ex}_{\text{values}})$ 
70:  kernel ( $i = 0 \dots product\_NNZ$ )
71:     $expansion\_index = \mathbf{C}.i[i]$ 
72:     $\mathbf{C}.i[i] = \mathbf{ex}_{\text{rows}}[expansion\_index]$ 
73:     $\mathbf{C}.x[i] = \mathbf{ex}_{\text{values}}[expansion\_index]$ 
74:  end kernel
75:  return C
76: end function

```

dering. The inner loop at line 28 loops over elements of a particular column of the \mathbf{A} matrix, while the outer loop (line 24) takes care of advancing onto the next columns. Note that col_skip is used to start the loop in the middle of a column, should that be required to equally balance the workloads. Also note that if \mathbf{A} is known to be rank deficient, the outer loop may need to advance multiple times, until reaching a non-empty column (such that $elemA < endA$ before entering the inner loop).

Once the expansion is calculated, the $\mathbf{ex}_{\text{rows}}$, $\mathbf{ex}_{\text{values}}$ pairs can be sorted while using the head flags as segment markers (note that the beginning of the first segment is implied and does not need to be explicitly represented). Finally, once the expansion is sorted, the boundaries of the elements and the columns can be easily spotted, and the number of nonzeros of the final product can be calculated, using the kernel at line 42. The variable $block_size$ refers to the size of the blocks of the EXCLUSIVESCAN kernel, which is selected at runtime to best fit the target GPU.

In the final compression stage (Algorithm 3), the storage for the product is calculated. In the first kernel of this phase, the expansion is scanned for column tails (changes in $\mathbf{ex}_{\text{cols}}$, line 55) and element tails (changes also in $\mathbf{ex}_{\text{rows}}$, line 56). The scan of the element flags gives the element index in the compressed matrix. Note that the reduction of these flags was already calculated in the previous stage (line 46), which could

be promoted to a scan to avoid recalculation, but storing the scans would require $O(\text{expansion size})$ memory and would be disadvantageous from both memory requirements and computational time standpoints.

Once the global index in the compressed matrix is known, indices of the final values of the elements in `ex_values` can be written (`C.i` is used as temporary storage), and `C.p` can be filled. Again, if the product is rank deficient, care needs to be taken: `C.p` might contain runs of multiple occurrences of the same index (including the zero index at the beginning), corresponding to the runs of empty columns.

Finally, the expansion values are summed up using segmented reduction, with `C.i` serving as tail flags, leaving the final values of the elements of the product at the tail positions in `ex_values` (line 69). The last kernel (line 70) merely copies these values to their compressed destinations in `C.p` and rewrites `C.i` by the corresponding row indices. Note that this kernel could be fused with the segmented reduction.

RESULTS

In this section, the timing results of sparse matrix multiplication performed using the proposed implementation¹ are compared with a similar state of the art implementation, CUSP 0.3.1 [1]. It was also compared to CSparse 1.2.0 [12], which runs on the CPU². Despite all effort, we were unable to find any existing OpenCL PSpGEMM implementations. The evaluation was performed by all-to-all multiplication of sparse matrices from The University of Florida Sparse Matrix Collection [11] and their transposes (for matrices which share a common dimension). This collection was chosen because it contains sparse matrices corresponding to a diverse set of problems, and as such it is suitable for testing of general purpose linear algebra implementations.

All the tests were performed on a computer with NVIDIA GeForce GTX 680 (3 GB RAM) and Tesla K40 (12 GB RAM), a pair of AMD Opteron 2360 SE CPUs running at 2.5 GHz and 16 GB of RAM. In both cases, the program was compiled as x64, and both CUDA and OpenCL used 64-bit pointers. The latest GPU drivers (version 344.48) were used. CUDA implementations were linked against CUDA 6.5 SDK libraries. During the tests, the computer was not running any time-consuming processes in the background. Each test was run at least ten times until cumulative time of at least 5 seconds was reached, and the average time was calculated in order to avoid measurement errors, especially on smaller matrices. Explicit CPU - GPU synchronization was always performed, using `cuCtxSynchronize()` or `clFinish()`, respectively. ECC was disabled on the Tesla GPU.

Our implementation works with the CSC format. The implementations working with CSR format had their matrices converted (transposed) accordingly. Recorded times do not include the conversion or data transfers. The benchmarked version of the proposed algorithm handles all the rank-deficient

¹The implementation of the proposed algorithm is available, at <https://sourceforge.net/projects/blockmatrix/>.

²CSparse is used as an orientative example, more efficient CPU implementations exist.

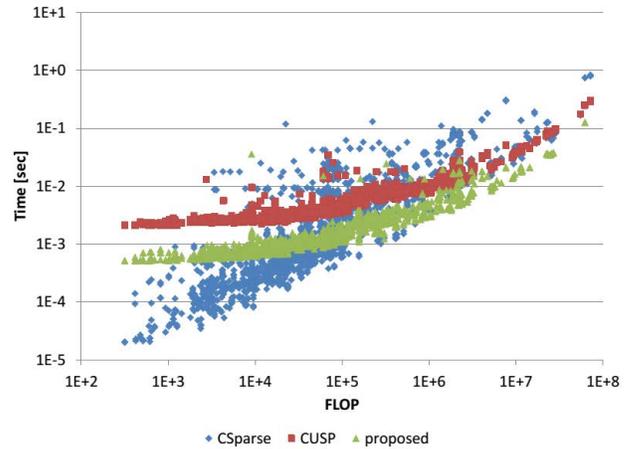


Figure 4. Performance scaling comparison on Tesla K40. Note that both axes are logarithmic.

cases described in the Section Implementation in a fully general way, without requiring prior detection or specialized kernels. The memory for the expansion and the product was allocated as outlined in Section Implementation, without any prior knowledge of the size of either. All the calculations were carried out in double precision.

Timing results for the all-to-all product benchmarks are on Fig. 4. Note that for very small matrices of less than ten thousand FLOP, CSparse is the fastest. For larger matrices, the proposed implementation takes over. Note that time of CSparse scales linearly with the number of FLOPs, as can be expected from a serial implementation of [16]. The times of the parallelized implementations grow slowly before the GPU gets saturated, then also scale approximately linearly. Least squares was employed to estimate the saturated costs to 27.7 ms/MFLOP for CSparse, 4.2 ms/MFLOP for CUSP and finally 3.0 ms/MFLOP for the proposed.

A more conventional comparison is presented in Table 1. This comparison was performed on the SNAP subset of the University of Florida Sparse Matrix Collection. It contains 9 different classes of matrices, a single matrix was chosen from each of them, much like evaluation in [17]. Each of the matrices was multiplied by itself (or in case of rectangular matrices, by its transpose). The proposed solution maintains the best times for most of the matrices, except for *roadNet - CA*, where the number of scalar products per element of the **B** matrix is very low, yielding high thread divergence in the proposed implementation. On smaller matrices such as *p2p - Gnutella31*, CUSP does not scale well and is slower despite the divergence. Reducing this divergence is the subject of the future work. Note that on *cit - Patents*, both the proposed and CUSP ran out of memory on GTX 680, and on *as - Skitter* there was not enough system memory to perform the multiplication even on the CPU. This is not a principal problem of the algorithm, rather it is an implementation issue. One would only need to add an extra parameter of how many columns of the **B** matrix should be processed at a time (corresponding to the same number of columns of

Table 1. Performance comparison on the SNAP subset, the best times are in bold (all times in seconds). The last two columns indicate relative speedup over CSparse and CUSP.

Matrix	nnz/row	FLOP	CSparse	GF GTX 680		Tesla K40				
				CUSP	ours	CUSP	ours	MFLOP/s	×CSp.	×CUSP
roadNet-CA	2.807	22.138 M	0.774	0.156	0.199	0.103	0.099	223.662	7.823	1.038
web-Google	5.571	91.665 M	5.312	0.447	0.433	0.315	0.249	368.356	21.347	1.265
email-Enron	10.020	72.510 M	1.150	0.360	0.271	0.247	0.173	418.961	6.645	1.429
amazon0312	7.987	42.368 M	1.659	0.209	0.241	0.141	0.123	344.389	13.488	1.148
ca-CondMat	8.081	5.899 M	0.140	0.035	0.027	0.024	0.015	394.591	9.347	1.575
p2p-Gnutella31	2.363	539.035 k	0.032	0.015	0.007	0.008	0.003	190.560	11.304	3.003
wiki-Vote	12.497	7.254 M	0.082	0.036	0.024	0.025	0.015	482.961	5.482	1.633
cit-Patents	4.376	95.457 M	13.414	<i>out of RAM</i>		0.497	0.446	214.127	30.089	1.114
as-Skitter	13.081	53.771 G	<i>out of RAM</i> ³							

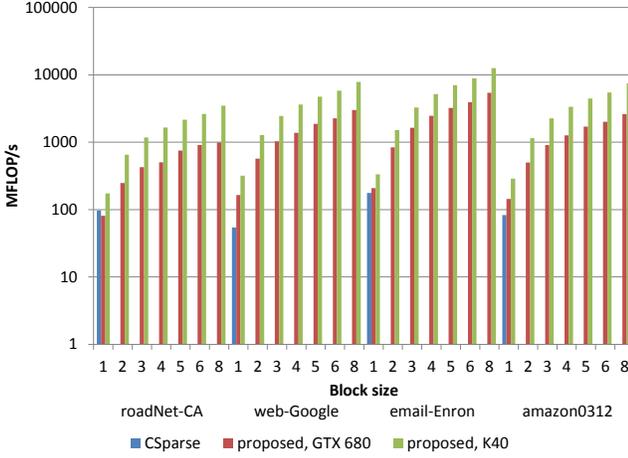


Figure 5. Performance scaling comparison of sparse block matrix multiplication on the first four matrices of SNAP.

the result), and the CPU would schedule the multiplication as several calls of the original algorithm.

For a synthetic benchmark of the sparse *block* matrix multiplication, the matrices from SNAP were used again. Each element was replaced by a dense block, while the block size was varied between the different tests. The results of this benchmark are on Fig. 5. As expected, the proposed implementation exhibits performance increase with increasing block sizes.

CONCLUSIONS AND FUTURE WORK

We presented a novel algorithm for sparse matrix multiplication and demonstrated its extension to sparse block matrices. The algorithm yields on average 329.7 MFLOP/s, outperforms CUSP by a factor of 1.53×, and outperforms CSparse running on a single CPU by a factor of 13.19×. The sparse block matrix multiplication exhibits further performance scaling with increasing block size, yielding up to 1.26 GFLOPS on Tesla K40 (*email – Enron*, 8 × 8 blocks). This makes it attractive in problems with block structure such as FEM, SLAM, BA or SfM. To improve the performance even more, multi-GPU or hybrid CPU-GPU extensions could be implemented. The implementation needs to be improved to handle large matrices by splitting the computation to bands, when the expansion does not fit in the GPU memory at once.

Currently, only constant block size compressed column format (CBC) is supported. This can be extended to variable block size compressed column (VBC), once addressing possible thread divergence problems. Also, to better integrate with the existing CPU pipelines which use the SSE instruction set, allocation of the product matrix with the proper memory alignment of the blocks needs to be solved.

ACKNOWLEDGEMENTS

The research leading to these results has received funding from the European Union, 7th Framework Programme grants 316564-IMPART and the IT4Innovations Centre of Excellence, grant n. CZ.1.05/1.1.00/02.0070, supported by Operational Programme Research and Development for Innovations funded by Structural Funds of the European Union and the state budget of the Czech Republic.

We gratefully acknowledge the support of NVIDIA Corporation with the donation of the GPUs used for this research.

REFERENCES

1. *CUSP: Generic parallel algorithms for sparse matrix and graph computations*. NVIDIA Corporation, 2009. <http://code.google.com/p/cusp-library/>.
2. *Intel Math Kernel Library. Reference Manual*. Intel Corporation, Santa Clara, USA, 2009. 630813-054US, <http://software.intel.com/intel-mkl/>.
3. Agarwal, R. C., Balle, S. M., Gustavson, F. G., Joshi, M., and Palkar, P. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development* 39, 5 (1995), 575–582.
4. Balay, S., Gropp, W. D., McInnes, L. C., and Smith, B. F. Efficient management of parallelism in object oriented numerical software libraries. In *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds., Birkhäuser Press (1997), 163–202.
5. Bell, N., Dalton, S., and Olson, L. N. Exposing fine-grained parallelism in algebraic multigrid methods. *SIAM Journal on Scientific Computing* 34, 4 (2012), C123–C152.

³Note that with only 16 GB of RAM, this matrix is too large even for CSparse: the product would take 26.4 GB.

6. Blelloch, G. E. *Vector models for data-parallel computing*, vol. 75. MIT press Cambridge, 1990.
7. Bowler, D., Miyazaki, T., and Gillan, M. Parallel sparse matrix multiplication for linear scaling electronic structure calculations. *Computer physics communications* 137, 2 (2001), 255–273.
8. Buluç, A., and Gilbert, J. R. Challenges and advances in parallel sparse matrix-matrix multiplication. In *Parallel Processing, 2008. ICPP'08. 37th International Conference on*, IEEE (2008), 503–510.
9. Choi, J., Walker, D. W., and Dongarra, J. J. Pumba: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers. *Concurrency: Practice and Experience* 6, 7 (1994), 543–570.
10. Dalton, S., Bell, N., and Olson, L. Optimizing sparse matrix-matrix multiplication for the gpu. *Technical Report* (2013).
11. Davis, T. The university of florida sparse matrix collection. In *NA digest*, Citeseer (1994).
12. Davis, T. A. *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. Society for Industrial and Applied Mathematics, 2006.
13. Dekel, E., Nassimi, D., and Sahni, S. Parallel matrix and graph algorithms. *SIAM Journal on computing* 10, 4 (1981), 657–675.
14. Fatahalian, K., Sugerman, J., and Hanrahan, P. Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, ACM (2004), 133–137.
15. Gunnels, J., Lin, C., Morrow, G., and Van De Geijn, R. A flexible class of parallel matrix multiplication algorithms. In *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International... and Symposium on Parallel and Distributed Processing 1998*, IEEE (1998), 110–116.
16. Gustavson, F. G. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software (TOMS)* 4, 3 (1978), 250–269.
17. Matam, K. K., Bharadwaj, S. R. K., and Kothapalli, K. Sparse matrix matrix multiplication on hybrid cpu+ gpu platforms. In *Proc. of 19th Annual International Conference on High Performance Computing (HiPC)*, Pune, India (2012).
18. Merrill, D., and Grimshaw, A. High performance and scalable radix sorting: A case study of implementing dynamic parallelism for gpu computing. *Parallel Processing Letters* 21, 02 (2011), 245–272.
19. Merrill, D. G., and Grimshaw, A. S. Revisiting sorting for gpgpu stream architectures. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, ACM (2010), 545–546.
20. Perl, Y., Itai, A., and Avni, H. Interpolation search - a log log n search. *Communications of the ACM* 21, 7 (1978), 550–553.
21. Polok, L., Ila, V., and Smrz, P. Cache efficient implementation for block matrix operations. In *Proceedings of the 21st High Performance Computing Symposia*, ACM (2013), 698–706.
22. Polok, L., Ila, V., Solony, M., Smrz, P., and Zemcik, P. Incremental block cholesky factorization for nonlinear least squares in robotics. In *Proceedings of the Robotics: Science and Systems 2013* (2013).
23. Polok, L., Solony, M., Ila, V., Zemcik, P., and Smrz, P. Efficient implementation for block matrix operations for nonlinear least squares problems in robotic applications. In *Proceedings of the IEEE International Conference on Robotics and Automation*, IEEE (2013).
24. Rennich, S. Leveraging matrix block structure in sparse matrix-vector multiplication. In *NVIDIA GPU Technology Conference* (2012).
25. Saravanan, C., Shao, Y., Baer, R., Ross, P. N., and Head-Gordon, M. Sparse matrix multiplications for linear scaling electronic structure calculations in an atom-centered basis set using multiatom blocks. *Journal of computational chemistry* 24, 5 (2003), 618–622.
26. Satish, N., Harris, M., and Garland, M. Designing efficient sorting algorithms for manycore gpus. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, IEEE (2009), 1–10.
27. Sengupta, S., Harris, M., Zhang, Y., and Owens, J. D. Scan primitives for gpu computing. In *Graphics Hardware*, vol. 2007 (2007), 97–106.
28. Shah, M., and Patel, V. An efficient sparse matrix multiplication for skewed matrix on gpu. In *High Performance Computing and Communication & 9th International Conference on Embedded Software and Systems (HPCC-ICESS)*, IEEE (2012), 1301–1306.
29. Zhang, F. *The Schur complement and its applications*, vol. 4. Springer, 2005.

ExaShark: A Scalable Hybrid Array Kit for Exascale Simulation

Imen Chakroun ExaScience
Life Lab, Belgium IMEC,
Leuven, Belgium

Tom Vander Aa ExaScience
Life Lab, Belgium IMEC,
Leuven, Belgium

Bruno De Fraine Vrije
Universiteit Brussels,
Brussels, Belgium

Tom Haber ExaScience Life
Lab, Belgium UHasselt,
Belgium

Roel Wuyts ExaScience Life
Lab, Belgium IMEC, Leuven,
Belgium DistriNet, KU
Leuven, Belgium

Wolfgang Demeuter Vrije
Universiteit Brussels,
Brussels, Belgium

ABSTRACT

Many problems in high-performance computing, such as stencil applications in iterative solvers or in particle-based simulations, have a need for regular distributed grids. Libraries offering such n-dimensional regular grids need to take advantage of the latest high-performance computing technologies and scale to exascale systems, but also need to be usable by non HPC experts. This paper presents ExaShark: a library for handling n-dimensional distributed data structures that strikes a balance between performance and usability. ExaShark is an open source middleware, offered as a library, targeted at reducing the increasing programming burden on heterogeneous current and future exascale architectures. It offers its users a global-array-like usability while its runtime can be configured to use shared memory threading techniques (Pthreads, OpenMP, TBB), inter-node distribution techniques (MPI, GPI), or combinations of both. ExaShark has been used to develop applications such as a Jacobian 2D heat simulation and advanced pipelined conjugate gradient solvers. These applications are used to demonstrate the performance and usability of ExaShark.

Author Keywords

N-dimensional grids, Partitioned Global Address Space, Exascale simulation, Middleware library, numerical solvers.

INTRODUCTION

High Performance Computing (HPC) architectures are expected to change dramatically in the next decade with the arrival of exascale computing, high performance computers that offer 1 exaFlop (10^{15}) of performance. Because of power and cooling constraints, large increases in individual core performance are not possible and as a result on-chip parallelism

is increasing rapidly. The expected hardware for an exascale machine node will therefore need to rely on massive parallelism both on-chip and off-chip, with a complex distributed hierarchy of resources. Programming a machine of such scale and complexity will be very hard unless some appropriate, workable layers of abstraction are introduced to bridge the gap between problem specification and efficient code at the cluster, node and chip levels.

Exascale machines will be mainly necessary for scientific and industrial simulations where scientists try to understand more complex phenomena by using simulations that run at ever higher resolutions and on ever longer time scales. One of the primary data structures in many of these scientific simulations is a regular multidimensional array. Indeed, a number of simulations can be modeled as time-discrete evolution on structured multidimensional array. Regular arrays are also at the core of many numerical algorithms.

Unfortunately, support for large-scale distributed multidimensional arrays is very limited. Such arrays generally must be manually mapped to one-dimensional arrays, with the user performing all the indexing logic. The problem is even worse in the parallel setting, since efficiently communicating subsets of an array that are not physically contiguous requires significant effort from the programmer. Some existing libraries (Global Array Toolkit [4], LibGeoDecomp [17], etc.) already offer support for n-dimensional regular grids. We show that ExaShark improves on the state of the art by providing a user-friendly, generic API without sacrificing performance. ExaShark takes advantage of the latest high-performance computing technologies, helping to scale to exascale systems.

This paper introduces ExaShark: a partitioned global address space (PGAS) library that provides a high-level, abstract interface for handling shared and distributed multidimensional arrays. It offers its users a global-arrays-like usability while its runtime can be configured to use any combination of shared memory threading techniques (Pthreads, OpenMP, TBB) and inter-node distribution techniques (MPI, GPI). Compared to similar general-purpose structured grid-

based application programming interfaces, ExaShark is enhanced with advanced features such as expression templates and operator overloading for global arrays, coupling of multiple grids and dynamic redistribution of the grid.

To validate our claims of performance and usability, the performance of ExaShark is evaluated on advanced state-of-the-art *pipelined conjugate gradient solvers*. We also compare ExaShark’s scalability with other widely used n-dimensional libraries on stencils-based applications such as the standard Jacobian 2D heat distribution benchmark.

The remainder of this paper is organized as follows. In Section 2, existing work dealing with multidimensional arrays is presented. Section 3 describes the ExaShark design focusing mainly on the communication layer, the data sharing patterns and the advanced features such as ghost regions. In Section 4, the experimental results are presented on two benchmarks: advanced numerical solvers and stencil codes. Some conclusions and perspectives of this work are drawn in Section 5.

RELATED WORK

This section presents existing work dealing with multidimensional arrays. We particularly focus on general-purpose libraries, not on specific libraries such as [8] and [16] that are specific to stencil computations which is only one optimization domain of ExaShark.

Despite the importance of multidimensional arrays to scientific applications, common programming languages such as C++ provide only limited support for such arrays (e.g. boost library).

However, a number of efforts aiming at creating reusable libraries to support scientists in implementing grids exist. The most commonly used PGAS array library is probably Global Arrays (GA) [4]. GA is implemented as a library with C and Fortran bindings, and more recently added Python and C++ interfaces (starting with the release 3.2). GA is built on top of the aggregate remote memory copy interface (ARMCI) [1], a low-level, one-sided communication runtime system. ARMCI forms GA’s portability layer, and when porting GA to a new platform, ARMCI must be implemented using that platform’s native communication and data management primitives. GA works with either MPI or TCGMSG message passing libraries for communication but it does not seem to support multi-threading. The fact that variables are always local to an MPI process and sharing them requires explicit communication between processes renders the “pure MPI” approach, without adding support for one-sided communication, unsustainable on future large-scale systems with growing numbers of cores and decreasing amount of memory per core.

In [17], LibGeoDecomp (Library for Geometric Decomposition codes) is presented. It is a generic C++ library with a high abstraction level that uses grids to express the computation in presence of high latency networks. It is limited to 2D and 3D simulations. LibGeoDecomp supports accelerators such as the Intel Xeon Phi coprocessor and GPU. Even though LibGeoDecomp provides a high-level C++ interface, it does not use specific mechanisms such as expression templates to speed up mathematical operations on the grids.

Our library, ExaShark, presented in the following section, is also enhanced with other features such as multiple grids coupling and dynamic redistribution of the grid. A comparison of different features is summarized in Table 1.

The following section describes the design of ExaShark. The focus is put on the communication layer, the data sharing patterns and the advanced features such as boundary conditions and expression templates over global arrays.

FRAMEWORK DESIGN

ExaShark is an open source middleware [6], offered as a library, targeted at reducing the increasing programming burden on heterogeneous current and future exascale architectures. ExaShark handles matrices that are physically distributed blockwise, either regularly or as the Cartesian product of irregular distributions on each axis. The access to the global array is performed through a logical indexing.

To define a global array, the programmer should define its coordinates which are the n dimensions of the structure and its data. The data can also be redistributed at run-time. Besides, the programmer can define halos/ghost regions in the global array (see Figure 1). Indeed, many applications based on regular grids need support for ghost cells. These regions are boundary data that are needed by one process to compute its inner part of the global array but which are remotely-held by other processes and need to be exchanged (updated) at each iteration. Because they are error prone and because they require knowledge expertise, ExaShark transparently manages these features for the programmer. Along with ghost cells, periodic boundary conditions are also supported.

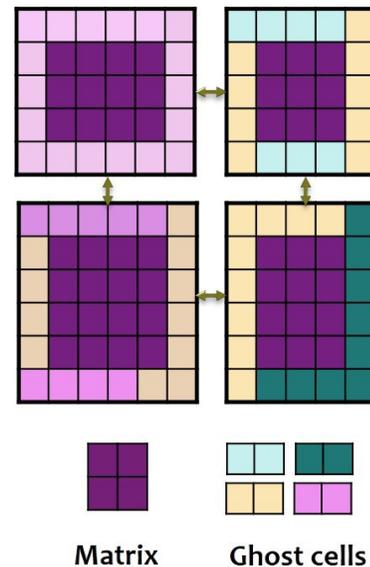


Figure 1. Different sized ghost regions configurations using ExaShark

Our major architectural drivers for a scalable structured grid-based library are efficiency, portability and ease of coding. ExaShark is portable since it is built upon widely used technologies such as MPI and C++ as a programming language. It provides simple coding via a global-arrays-like interface

Desing Flaws	ExaShark	Global Array	LibGeoDecomp
Programming language	C++	Pyhton/Fortran/C++	C++
Ghost regions	Yes	Yes	Yes
Communication layer	Hybrid (MPI + OpenMP/Pthreads)	MPI/TCGMSG	MPI/OpenMP
Expression templates for ndim arrays	Yes	Yes	No
Accelerator support (XeonPhi/GPU)	XeonPhi	No	Yes
Domain redistribution	Yes	No	Yes
Stencil operations	Yes	No	Yes
Vector operations (SPMV, dot products..)	Yes	Yes	No
Coupling multiple grids	Yes	No	No

Table 1. Comparing features of ExaShark with similar libraries

which offers template-based functions (dot products, matrix multiplications, unary expressions). The functionalities offered by ExaShark are efficient since they use asynchronous and specific communication patterns. For example, let us consider the update operation which fills in ghost cells with the visible data residing on neighboring processors. This operation usually induces latency. In order to hide this latency, ExaShark provides an asynchronous version where the data exchanges are overlapped with the update of the inner region of each process.

Data distribution

ExaShark allows the programmer to control regular and irregular data distributions of the global array. Indeed, ExaShark is based on the PGAS parallel programming model which is convenient for expressing algorithms with large and random data access. Each process is assumed to have fast access to a portion of each distributed matrix, and slower access to the remainder. These access time differences define the data as being either local or remote, respectively. This locality information can be exploited at each iteration of the computation. For example, the user can inquire about what data portion is held by a given process or which process owns a particular array element, identify inner and outer regions of the grid, etc.

ExaShark enables user-configurable wide ghost zones: a ghost zone with width k . It also allows ghost cell widths to be arbitrarily set in each dimension, as sketched in Figure 1, thereby allowing programmers to improve performance by combining multiple fields into one global array and using multiple time steps between ghost cell updates.

Communication layer

As quoted in Section 2, ExaShark supports a plethora of lower level programming models and libraries with the aim of being adequate to exascale systems which are highly heterogeneous. Application developers may use pure MPI and/or hybrid MPI + OpenMP threads to target coarse and medium-grained parallelism. Work on integrating GASPI/GPI (Global Address Space Programming Interface) [11] is ongoing and

should provide an even faster, more efficient, and more scalable communication between processes.

One of the ExaShark’s efficiency keys is that it uses asynchronous communications and ensures overlapping communication and computation whenever possible. Several types of communications are used according to the computation performed. For instance, for the update of the ghost regions geometric communication is needed, while one sided communication is used for the get and put routines and collective operations are performed for the reductions.

Expression templates over global arrays

ExaShark offers many high-level functions traditionally associated with arrays, eliminating the need for programmers to write these functions themselves. Examples are basic mathematical operators, unary functions, standard global arrays operations such as dot products and matrix vector multiplication, and so on. These functionalities are implemented using expression templates [18].

In Figure 2, we show how a basic code (left) of the conjugate gradient solver which uses mathematical operations over vectors is rewritten using ExaShark (right). The goal of this algorithm is to give an approximate solution of the equation $Ax = b$ where the A is the coefficient matrix, x is the solution vector that is returned and b is the input vector. The first instruction of the Exashark code (left) defines the global domain size (in this case $n*n$). The second instruction defines two global arrays which corresponds to the x and b vectors in the pseudo code of CG (left). Here the matrix A is not allocated since it is implicitly assembled using stencils. Indeed, the information about the underlying matrix is available only through matrix vector computations which is probed approximately without forming and storing the elements and therefore do not have access to a fully assembled matrix [15].

By using expression templates, ExaShark incorporates delayed expressions to reduce temporary memory allocation. For example, consider line 7 from the code sample in Figure 2. A naive implementation of the operation $x = x + \alpha p$ where alpha is a scalar and x and p are global arrays would have `operator+` and `operator*` overloaded and return ar-

rays. Then, the considered expression would mean creating a temporary for ap then another temporary for $x+$ that first temporary, then assigning that to x . Even with the return value optimization this will allocate memory at least twice. Expression templates delay evaluation so the expression essentially generates at compile time a new array constructor. It is as if this constructor takes a scalar and two arrays by reference; it allocates the necessary memory and then performs the computation. Thus only one memory allocation is performed.

<pre> 1. $r = b - Ax^{(0)}$ 2. $\rho_0 = \ r\ _2$ 3. $k = 0, p = r, x = x^{(0)}$ 4. while $\rho \geq \epsilon$ and $k < k_{max}$ 5. $w = Ap$ 6. $\alpha = \rho_k^2 / (p^T w)$ 7. $x = x + \alpha p$ 8. $r = r - \alpha w$ 9. $\rho_{k+1} = \ r\ _2$ 10. $\beta = \rho_{k+1}^2 / \rho_k^2$ 11. $p = r + \beta p$ 12. $k = k + 1$ </pre>	<pre> coords size = {n,n}; Domain d(...,size); GlobalArrayD x (...),b(...); 1. r = b - A * x; 2. rho = norm2(r); 3. p = r, k = 0; 4. while (rho >= tol) && (k<maxit) { 5. w = A * p; 6. alpha = rho*rho / dot(p,w); 7. x = x + alpha * p; 8. r = r - alpha * w; rho_old = rho; 9. rho = norm2(r); 10. beta=rho*rho/ (rho_old*rho_old); 11. p = r + beta * p; 12. k = k+1; } </pre>
---	--

Figure 2. Pseudo code (left) vs running code using ExaShark (right) of the conjugate gradient solver

Inter-operability and interfacing with external software

ExaShark can interface with external libraries when needed by the user’s application. For example, developers can use the functionality of the Intel’s Math Kernel Library (MKL) for optimized math routines [13] or PETSc (Portable, Extensible Toolkit for Scientific Computation) [7] for advanced numerical methods for partial differential equations and sparse matrix computations. Work on integrating stencils compilers such as Patus [10] into ExaShark for optimizing stencil code implementations is also ongoing.

The PETSc solvers, for example, can be called within an Exashark application to solve PDEs that require solving large-scale, sparse nonlinear systems of equations. One important issue related to inter-operability is how to convert the data structures before calling the PETSc solvers, and how to convert the data structures of PETSc back after calling the PETSc solvers. We believe that one of the most efficient ways to exchange data structures between Exashark and PETSc without inducing data copying overheads, is to use the `V ecGetArray()` and `V ecRestoreArray()` routines of PETSc. `V ecGetArray()`, indeed, returns a pointer to a contiguous array that contains a processor’s portion of the vector data. After calling the PETSc solvers, using `V ecGetArray()` will zeros out the pointer unless vector’s data are not stored in a contiguous array. In this case, this routine will copy the data back into the underlying vector data structure from the array obtained with `V ecGetArray()`.

For integration with Patus, the idea is to improve the performance of stencil operations at the node-level, while ExaShark orchestrates the distribution of data and computation across different nodes and manages the communication between them. Here as well, the data conversion and memory alignment for the structures used by both software should be carefully taken into account. Our plan is to use the stencil-embedded definition provided by Patus which allows to exchange only pointers rather than performing data copies.

Generally speaking, it is not hard to integrate with external libraries because internally ExaShark stores its data in the common row-wise pattern. It is therefore sufficient in many cases to exchange pointers. If other data formats are used then integration is still possible but may incur the penalty of copying data from and/or to the external representation.

VALIDATION

In this section we present the experimental results obtained with ExaShark on two benchmarks and compare them with the results of the libraries presented in Section We also evaluate the usability of each of the considered libraries using software design metrics.

Hardware platform

The experimental validation was carried out on the Anselm supercomputer cluster [2] which consists of 209 compute nodes, totaling 3344 compute cores with 15TB RAM and giving over 94 Tflop/s theoretical peak performance. Each node is a x86-64 computer, equipped with 16 cores, at least 64GB RAM, and 500GB hard drive. Nodes are interconnected by a fully non-blocking fat-tree Infiniband network and equipped with Intel Sandy Bridge processors.

Benchmarks

Two applications are considered as benchmarks for measuring the performance of ExaShark. The first one is the conjugate gradient method (CG) [12] which is an algorithm for the numerical solution of particular systems of linear equations. The second one is a heat distribution simulation example which is a five point stencil application. The aim of the first experiment is to evaluate the added value of asynchronous and hybrid communication techniques supported by ExaShark. The second is to measure the performance of ExaShark in comparison to other similar libraries namely GA and LibGeoDecomp.

Conjugate Gradient method

Iterative methods are an efficient way to obtain a good numerical approximation to the solution of $Ax = b$ when the *matrix* A is large and sparse. The CG method is a widely used iterative method for solving such systems when the *matrix* A is symmetric and positive definite. Generalizations of CG exist for non-symmetric problems. A pseudo code of the conjugate gradient method is given in Figure 2.

We run three experiments for the CG solver using ExaShark but with different communication protocols: the first scenario is a CG implementation that relies on asynchronous communication and for which hybrid asynchronous MPI/OpenMP is used. In the second scenario, the same implementation of

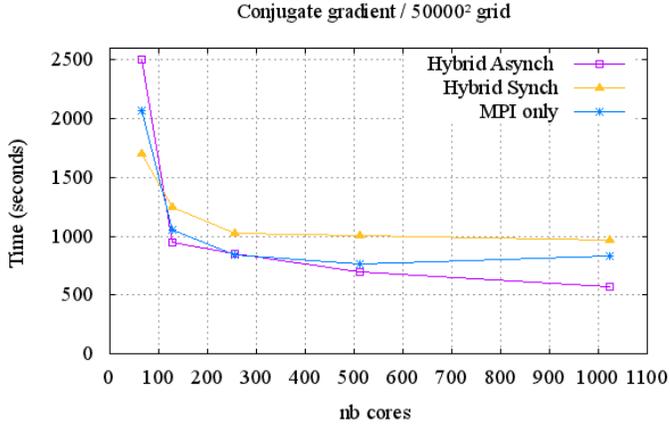


Figure 3. Results for the conjugate gradient method with ExaShark

CG is run with only MPI. The last scenario corresponds to an implementation of CG where no asynchronous communication is considered. For this later scenario hybrid synchronous MPI/OpenMP is used.

The results, reported in Figure 3 show that using both intra and inner nodes asynchronous communication techniques are on average 32.17% and 19.4% better than synchronous communication and message passing interface mechanisms respectively. For example, resolving the same problem instance using 1024 cores would take 576 seconds with the hybrid asynchronous version of CG while it would last 971 seconds using the hybrid synchronous communication and 835 seconds with only asynchronous MPI. Results show also that the speedup with asynchronous hybrid communication protocols scales better than the other communication techniques. The results are machine and preconditioner-dependent.

Heat distribution using the Jacobi iteration

Many scientific applications use iterative finite-difference techniques that sweep over a spatial grid, performing nearest neighbor computations called stencils [14]. In a stencil operation, each point in a multidimensional grid is updated with weighted contributions from a subset of its neighbors, locally solving a discretized version of a partial differential equation for that data element.

Jacobi [9] is a popular algorithm for solving Laplace’s differential equation on a square domain, regularly discretized. The idea is the following: let us consider a 2D array of particles, each with an initial value of temperature. Each particle is in contact with a fixed value of neighboring particles which impact its temperature. The Laplace’s equation is solved for all particles to determine their temperature as the average of the four neighboring particles. To do so, a number of iterations are performed over the data to recompute average temperatures repeatedly, and the values gradually converge to a finer solution until the desired accuracy is reached.

The results described in Figure 4 show that for the 2D heat distribution simulation, ExaShark is on average 59% faster than the Global Array Toolkit for the 32768² grid and 52%

for the 65536² grid. We can also see that ExaShark scales better than GA. On the other hand, LibGeoDecomp is on average 39% faster than ExaShark for the 32768² grid but we could not get it to work for bigger simulations namely for sizes larger than 65536². This constraint is a significant problem for exascale numerical simulations. Indeed, our simulations with LibGeoDecomp crashed for the 2D heat distribution simulation of a 65536² grid because it consumes more than 64GB of data, which is the maximum allowed memory size on our testing cluster.

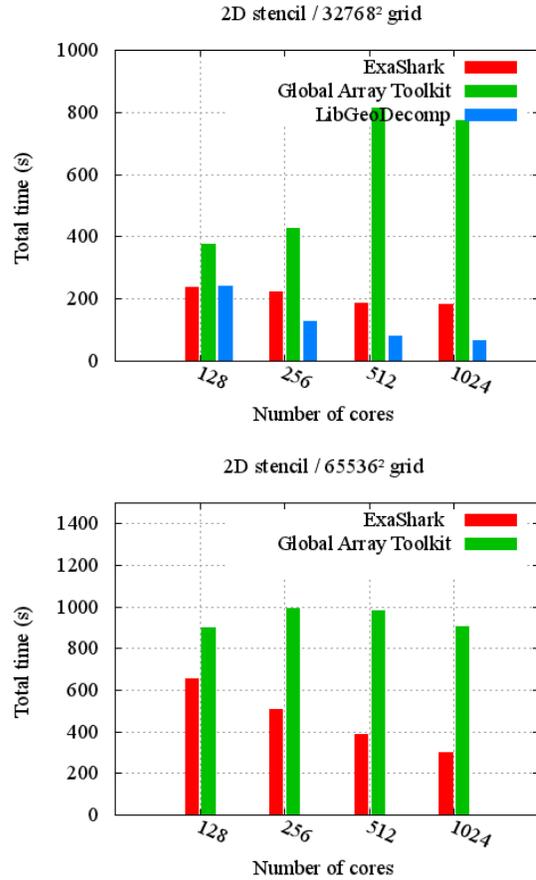


Figure 4. Comparing the performance of ExaShark, GA and LibGeoDecomp on the 2D heat distribution problem

As quoted in Section 2, GA seem to support multi-threading. The fact that variables are always local to an MPI process and sharing them requires explicit communication between processes renders the “pure MPI” approach unsustainable on future large-scale systems with growing numbers of cores and decreasing amount of memory per core.

Figure 5 show the results of weak scaling for all considered libraries. The results show that LibGeoDecomp scales very well compared to ExaShark and GA as long as the grid size was smaller than 65536² on our 64GB compute node. The scalability obtained with GA and ExaShark can be explained by the nature of the benchmark itself. Indeed, as most stencil codes, the heat dissipation problem performs repeated sweeps through data structures that are typically much larger than the

data caches of modern microprocessors. As a result, these computations generally produce high memory traffic for relatively little computation, causing performance to be bound by memory throughput rather than floating-point operations.

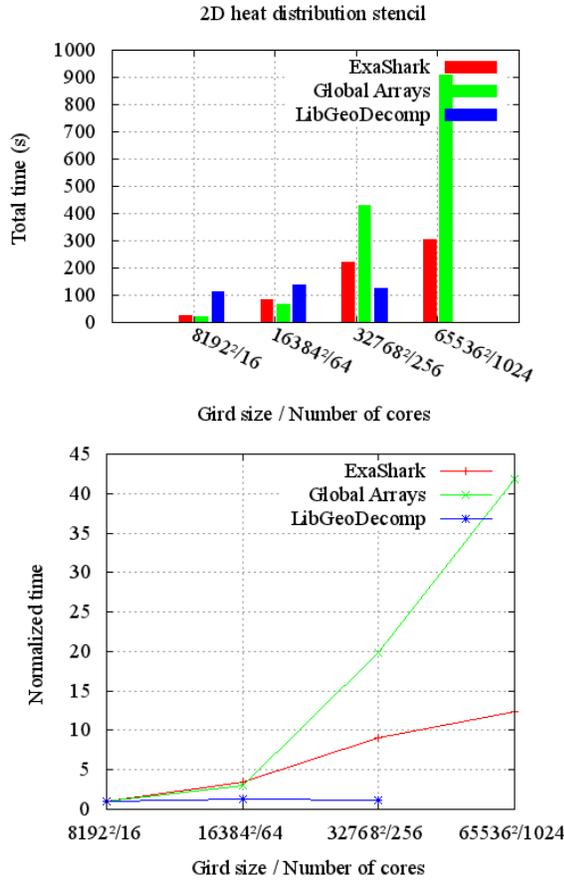


Figure 5. Analyzing weak scaling of ExaShark, GA and LibGeoDecomp on the 2D heat distribution problem.

Usability

Because simple code is always better than complex code, we tried to evaluate the usability of each of the three libraries considered in this work: ExaShark, GA and LibGeoDecomp. To do so, we used the “inCode Helium”[5] software which is a quality assessment tool for Java, C++ and C programs. InCode detects design flaws and helps to understand the causes of quality problems on the level of code and design.

From the code point of view, InCode computes software metrics for a given program ranging from basic size and complexity metrics to the more advanced metrics. In Table 2, we report two metrics: the number of classes and the number of lines. We can see, according to these metrics, that ExaShark has the least number of lines of codes and classes which is an indication of an easier use of the library.

From the design point of view, InCode detects design anti-patterns that are most commonly encountered in software

Metrics	ExaShark	Global Array	LibGeoDecomp
Number of Lines of code	8881	501553	65925
Number of classes	12	464	535

Table 2. Number of lines and classes of Exashark, GA and Libgeodecomp

projects. These anti-patterns cover all the essential design aspects: complexity, encapsulation, coupling, cohesion and inheritance. We ran InCode for GA, ExaShark and LibGeoDecomp and report the corresponding design flaws in Table 3. InCode shows that ExaShark has no design flaws whereas both other libraries have. Some of these design flaws, such as “data clumps” and “feature envy”, affect the understanding of the operations provided by a library and consequently its usability. An explanation of the different design flaws reported in Table 3 is given below:

- Data Class: refers to a class with an interface that exposes data members, instead of providing any substantial functionality. This means that related data and behavior are not in the same scope, which indicates a poor data-functionality proximity.
- Code Duplication (internal/external): refers to groups of operations which contain identical or slightly adapted code fragments. Duplicated code multiplies the maintenance effort, including the management of changes and bug-fixes. Moreover, the code base gets bloated.
- Data Clumps: are large groups of parameters that appear together in the signature of many operations. This hampers the understanding of operations with a data clump.
- Feature Envy: refers to an operation that manipulates a lot of data external to its definition scope. This is a strong indication that the affected operation was probably misplaced and that it could be moved to the scope in which the ‘en-vied’ data resides.
- Schizophrenic Class: describes a class with a large and non-cohesive interface. Such classes represent more than a single key abstraction and this affects the ability to understand and change in isolation the various abstractions embedded in the class.
- Tradition Breaker: refers to a class that breaks the interface inherited from a base class or an interface. A class can do this by reducing the visibility of the services published by the base class by means of private/protected inheritance (in C++).

CONCLUSION AND FUTURE WORK

In this paper we introduced ExaShark: a PGAS-based library that provides a high-level interface for handling shared and distributed multidimensional arrays. ExaShark supports hybrid shared-memory threading technologies such as Pthreads and OpenMP as well as distributed-memory technologies such as MPI. Compared to similar general-purpose structured grid-based libraries, ExaShark is enhanced with advanced features such as templates and operator overloading for global

Design Flaws	ExaShark	Global Array	LibGeoDecomp
Internal Duplication	0	280	2
External Duplication	0	729	5
Feature Envy	0	122	0
Data Clumps	0	1153	67
Data Class	0	249	22
Tradition Breaker	0	0	8
Schizophrenic Class	0	0	2

Table 3. Design flaws for Exashark, GA and LibGeoDecomp

arrays, multiple grids coupling and dynamic redistribution of the grid. Two benchmarks have been used for the experimental validation of ExaShark: advanced pipelined conjugate gradient solvers and stencils-based applications such as a 2D heat simulation. The first application was used to highlight the benefits of using asynchronous and hybrid communication techniques within the library. The second to demonstrate the performance of ExaShark comparing to other similar libraries. The results shows that using both intra- and inner-nodes asynchronous communication techniques are 32.17 % and 19.4% better than synchronous communication and message passing interface mechanisms respectively. Compared to GA, ExaShark is on average 55% faster and scales better for the 2D heat distribution simulation. LibGeoDecomp is faster than ExaShark but we could not get it to work for bigger simulations namely for sizes greater than 65536^2 . This constraint is a significant problem for exascale numerical simulations. We also tried to evaluate the usability of ExaShark using the “inCode helium” tool which identified design flaws in GA and LibGeoDecomp but not in ExaShark. Some of these design flaws, such as “data clumps” and “feature envy”, affect the understanding of the operations provided by a library and consequently its usability.

As future work, we are working on a GASPI/GPI (Global Address Space Programming Interface) integration within ExaShark [11]. For optimizing stencil code implementations, stencils compilers like PATUS [10] will be included into ExaShark.

ACKNOWLEDGMENT

This work is part of the European project EXA2CT [3]. EXA2CT aims at discovering solver algorithms that can scale to the huge numbers of nodes at exascale, developing an exascale programming model that is usable by application developers and offering these developments to the wider community in open-source proto-applications as a basis to develop real exascale applications.

We thank Dr. Pascal Costanza from the ExaScience Lab Belgium and Intel Health & Life Sciences for his help.

REFERENCES

- Aggregate remote memory copy interface. <http://hpc.pnl.gov/armci/documentation.htm>.

- Anselm supercomputer cluster. [https://docs.it4i.cz/anselm/discretionary\(-\){}{cluster\discretionary\(-\){}{documentation](https://docs.it4i.cz/anselm/discretionary(-){}{cluster\discretionary(-){}{documentation).
- Exascale algorithms and advanced computational techniques. <http://www.exa2ct.eu/>.
- The global array toolkit. <http://hpc.pnl.gov/globalarrays>.
- Incode helium. <https://www.intooitus.com/products/incode>.
- Scalable hybrid array kit. <https://github.com/ExaScience/shark>.
- Balay, S., Abhyankar, S., Adams, M. F., Brown, J., Brune, P., Buschelman, K., Eijkhout, V., Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L. C., Rupp, K., Smith, B. F., and Zhang, H. PETSc users manual. Tech. Rep. ANL-95/11 - Revision 3.5, Argonne National Laboratory, 2014.
- Bianco, M., and Varetto, U. A generic library for stencil computations. *CoRR abs/1207.1746* (2012).
- Cecilia, J. M., García, J. M., and Ujaldon, M. CUDA 2D Stencil Computations for the Jacobi Method. In *PARA (1)*, K. Jónasson, Ed., vol. 7133 of *Lecture Notes in Computer Science*, Springer (2010), 173–183.
- Christen, M., Schenk, O., and Burkhart, H. PatuS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium, IPDPS '11*, IEEE Computer Society (Washington, DC, USA, 2011), 676–687.
- Grünwald, D., and Simmendinger, C. The GASPI API specification and its implementation GPI 2.0.
- Hestenes, M. R., and Stiefel, E. Methods of conjugate gradients for solving linear systems.
- Intel. Using intel math kernel library for matrix multiplication. https://software.intel.com/en-us/mkl_11.2_tut_c_pdf.
- Kamil, S., Chan, C., Olikek, L., Shalf, J., and Williams, S. An auto-tuning framework for parallel multicore stencil computations. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, IEEE (2010), 1–12.
- Knoll, D. A., and Keyes, D. E. Jacobian-free Newton–Krylov methods: a survey of approaches and applications. *Journal of Computational Physics* 193, 2 (2004), 357–397.
- Lengauer, C., Apel, S., Bolten, M., Größlinger, A., Hannig, F., Köstler, H., Rude, U., Teich, J., Grebhahn, A., Kronawitter, S., Kuckuk, S., Rittich, H., and Schmitt, C. ExaStencils: Advanced Stencil-Code Engineering.

17. Schäfer, A., and Fey, D. Libgeodecomp: A Grid-Enabled Library for Geometric Decomposition Codes. In *PVM/MPI*, A. L. Lastovetsky, M. T. Kechadi, and J. Dongarra, Eds., vol. 5205 of *Lecture Notes in Computer Science*, Springer (2008), 285–294.
18. Veldhuizen, T. Expression templates. In *C++ Report*, vol. 7 (June 1995), 26–31.

A Load Balancing Parallel Method for Frequent Pattern Mining on Multi-core Cluster

Lan Vu, Gita Alaghband
University of Colorado Denver,
{lan.vu, gita.alaghband}@ucdenver.edu

ABSTRACT

In this paper, we present a new parallel method named SDFEM that enables frequent pattern mining (FPM) on cluster with multiple multi-core compute nodes to provide high performance. SDFEM is distinguished from previous parallel FPM works due to incorporating three advanced features to provide high mining performance for large-scale data analytic applications. First, SDFEM combines both shared memory and distributed memory computational models to leverage benefits of shared memory within a node in cluster. Second, it employs a multi-strategy load balancing approach to address the most challenging issue of parallel FPM to balance the mining workload among all cores of the cluster. Finally, its self-adaptive mining solution with the capability of dynamically adjusting to the characteristics of the database to perform efficiently on different data types either sparse or dense. For performance evaluation, we implement SDFEM using a hybrid model of OpenMP and MPI in which OpenMP is for the shared memory model and MPI is for message passing. SDFEM has been tested on a cluster of multiple 12-core shared memory compute nodes. Our experimental results on real databases show that performance of SDFEM is up to 329.5% faster than the parallel FPM approach that uses only distributed memory model with message passing (i.e. using pure MPI). In addition, SDFEM can achieve up to 45.4 – 64.8 speedup on 120 cores (i.e. 10 compute nodes and 12 cores per node).

Author Keywords

Frequent pattern mining; multi-core cluster; high performance computing; load balancing; database.

HPC 2015, April 12 - 15, 2015, Alexandria, VA, USA

© 2015 Society for Modeling & Simulation International (SCS)

INTRODUCTION

Frequent pattern mining (FPM) is a crucial component of data mining used to find various types of relationships among variables in large databases such as associations [1], correlations [2], causality [3], sequential patterns [4], episodes [5] and partial periodicity [6]. It has many practical applications such as market analysis, biomedical and computational biology, web mining, decision support, telecommunications alarm diagnosis and prediction, and network intrusion detection [7, 8]. With growth of big data in numerous fields such as business, social media, life science, medicine, etc., applying high performance computing (HPC) using large cluster computers for FPM are essential. These machines provide massive computing and memory resources making them ideal for big data analysis. Development of high performance methods for FPM requires platform-specific design to efficiently leverage the specific platform's powerful resources.

Motivation

Our study of parallelizing FPM for large-scale data mining applications on multi-core clusters addresses three critical problems that have not been thoroughly investigated in previous studies. Solving these problems are challenging because parallel FPM usually involves multiple reduction steps, large synchronization cost and unpredictable workload for load balancing.

First, most current HPC machines are clusters consisting of many multi-core nodes whose memory is shared among cores within a node but is not directly accessible from other nodes. Recent studies have shown that a hybrid parallel programming method that applies both shared and distributed memory programming models for cluster architecture delivers better performance than parallel methods using only distributed memory model [9–13]. However, most FPM methods designed for cluster use “shared nothing” parallel model; communication among parallel processes is, therefore, done by message passing [8, 14–27]. As a result, benefits of shared memory available within each node are ignored.

Second, load balancing is highly critical for parallel FPM in cluster computing environment. FPM incurs high message passing communication cost due to large variation and amount of data communications; making load balancing a great challenge. Irregular and imbalanced computation loads may result in sharp degradation of the overall performance [19]. An efficient workload balancing solution is critical for FPM scalability on cluster architectures.

Finally, in our prior study, we have developed a sequential frequent pattern mining method that can dynamically adjust to the characteristics of the database at runtime as the pattern mining proceeds and outperforms most well-known sequential methods both on sparse and dense databases [28–31]. Applying this mining approach for parallel FPM is essential to provide high performance for this mining task on sparse and dense data.

Contribution

We present a novel parallel FPM algorithm, SDFEM, to address the above-mentioned issues. SDFEM efficiently adapts to the architecture of multi-core clusters and maximizes utilization of the available computing resources. SDFEM is distinguished from prior work due to the following features: (1) exploits the use of shared memory within a node of the cluster, (2) applies multi-level load balancing and (3) uses a self-adaptive mining approach based on data characteristics. Highlights of our contributions include:

- 1) SDFEM algorithm, a hybrid parallel method utilizing both shared and distributed memory programming models that performs communication within-node via shared memory and between-node via message passing. Using shared memory inter-process communication cuts down the communication cost which is quite high for message passing communication among parallel processes and reduces load balancing overhead. SDFEM also employs the mining based on data characteristics approach for faster FPM performance on both sparse and dense data.
- 2) A multi-level load balancing approach that uses four different strategies: dynamic job scheduling and work sharing for load balancing among cores within a node, and cyclic job scheduling and work stealing for load balancing among nodes in the cluster. This load balancing method is designed based on the data communication features of the hybrid mining model to minimize the overhead of the load balancer as well as to enhance the scalability of FPM.
- 3) Implementation of the algorithms using MPI (message passing interface) and OpenMP (shared memory), and performance evaluation using real-world datasets to demonstrate the efficiency of the proposed method.

BACKGROUND

Problem Statement

The FPM problem is defined as follows: Let $I = \{i_1, i_2, \dots, i_n\}$ be the set of all distinct items in the transactional database D . The *support* of an *itemset* α (a set of items) is the percentage of transactions containing α in D . A k -*itemset* α , which consists of k items from I , is frequent if α 's *support* is larger or equal to *minsup*, where *minsup* is a user-specified minimum support threshold. Given a database D and a *minsup*, FPM searches for the complete set of frequent itemsets in D . For example, given the database in Table 1 and *minsup*=20%, the frequent 1-itemsets include a, b, c, d and e while f is infrequent because the *support* of f is only 11%. Similarly, $ab, ac, ad, ae, bc, bd, cd, ce, de$ are frequent 2-itemsets and abc, abd, ace, ade are the frequent 3-itemsets.

Table 1. Sample dataset with *minsup* = 20%

Transaction ID	Items	Sorted Frequent Items
1	b,d,a	a,b,d
2	c,b,d	b,c,d
3	c,d,a,e	a,c,d,e
4	d,a,e	a,d,e
5	c,b,a	a,b,c
6	c,b,a	a,b,c
7	f	
8	b,d,a	a,b,d
9	c,b,a,e,f	a,b,c,e

Related Works

A number of parallel methods have been developed for distributed memory systems. However, these methods do not take advantage of the shared memory within a node because they apply distributed memory computing model and eliminate the fact that cores within a node share same memory space. Most methods succeed in reducing data communication and increasing data independence among parallel processes but suffer from load imbalance since their load balancing strategy is heavily based on data partitioning. As a result, they may not scale well in cases like mining with small *minsup* which results in very large number of frequent patterns (the smaller the *minsup*, the larger the number of produced frequent patterns).

Pramudiono et al. [32] proposed a parallel shared nothing FPM method based on FP-growth, a well-known sequential FPM method. It partitions data equally among nodes to construct local FP-trees and deploys a model similar to MapReduce to map conditional pattern bases from a send process to a receive process. The method utilizes a characteristic called “path depth” to determine the size limit of conditional pattern bases to balance the workload among processes. Work balance is maintained by randomly selecting a process for work distribution. This approach is efficient when a good selection of path depth is made. The load balancing strategy with random selection of process is similar to one of load balance strategies in our proposed method. Yu et al. presented a parallel FPM solution for a

homogeneous PC cluster [19] based on FP-growth [25]. It uses a sampling technique for load balancing. However, they used synthetic data for experiments and reported poor performance. The method proposed by Tanbeer et al. [18], also based on FP-growth, required a simple database scan using a Parallel Pattern Tree. This study did not describe a load balancing strategy.

For mining dense databases, Suchahyo et. al [33] proposed a parallel method based on a sequential FPM method called Eclat [34]. The database is partitioned into projections, one for each item. Each projection, whose size depends on data characteristics, is stored on the local disk of a node in the cluster. Load balancing is done by distributing the projections to nodes in a round-robin fashion. All nodes follow the same order for deciding the destination node for the conditioned pattern bases, so there is a potential for blocking [32]. Similar to the method by Pramudiono [32], O'zkural et al. presented a parallel solution using vertical data layout [27] and applied a top-down data partitioning scheme in such a way that entire database could be divided into parts with some replications so that they could be mined independently. The data were partitioned to minimize replications and maintain storage balance and computational load. Similar to the FP-growth based methods [18], [25], [32], the benefit of shared memory in multi-core clusters was ignored. Nevertheless, for dense data, they obtained good performance

Another well-known sequential method is Apriori [1]. DPA (Distributed Parallel Apriori) proposed by Yu et al. [19], is one of the few FPM methods that parallelize Apriori. Because DPA uses a breadth first mining, it is easier to maintain load balancing than in methods using depth first mining strategy. However, this approach requires multiple database scans and suffers from large synchronization overhead because of multiple iterations of the mining loop. Furthermore, Apriori usually has lower performance than most other mining methods; its parallel version (DPA) exhibits a similar poor performance level.

In summary, most existing methods supply their own strategies for data partitioning and job scheduling to balance workload and minimize communication among parallel processes. However, none of them considers the situation where load balance cannot be obtained via work partitioning, particularly when mining with very small *minsup*. This together with the three issues described in previous section motivate the development of a new parallel FPM method.

SDFEM ALGORITHM

Overview

SDFEM performs FPM by deploying a group of parallel processes and mapping each process to a multi-core node. Each process creates a group of shared memory threads, mapping each thread to a core. Threads of the same process

collaborate to construct the process's projected XFP-tree. The XFP-tree is a new extension of the FP-tree [29]. The XFP-tree is a prefix tree storing compact mining data in memory. On each process, the XFP-tree is built from transactions projected to a group of items assigned to its process. The XFP-tree data structure uses shared memory and allows for some node replication but keeps the total count constant. This enhances parallelism among threads in that construction of the XFP-tree requires minimal synchronisation among threads and it is shared among the shared memory threads. Each thread uses the XFP-tree to generate its own frequent patterns [29], with each thread applying both FP-tree and bit vectors. SDFEM finally aggregates frequent patterns generated by all threads for the final output. SDFEM combines features of both distributed memory and shared memory programming models where between-node communication is done using message passing and within-node communication is done using shared memory. Figure 1 illustrates the execution model of SDFEM.

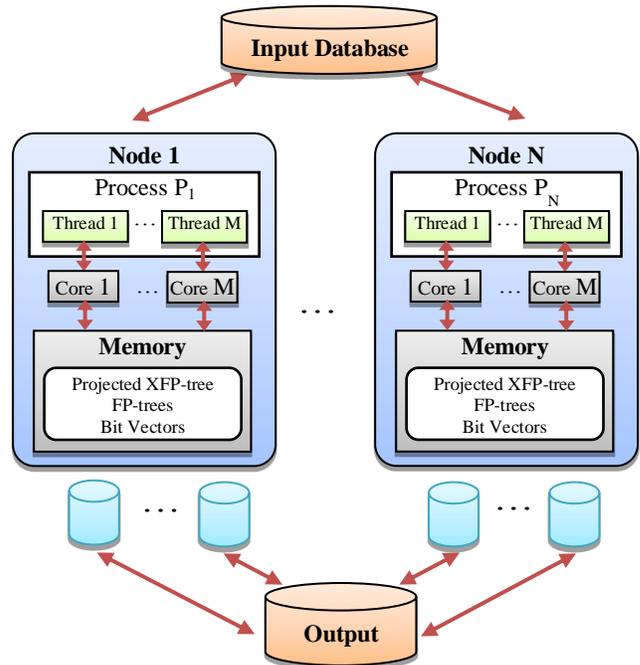


Figure 1: Overview execution model of SDFEM

SDFEM model can significantly reduce the overhead of data communication and allow more efficient load balancing. We develop a multi-level load balancing method for SDFEM using four different techniques to increase the CPU utilization and enhance performance. SDFEM performs FPM in two stages: parallel projected XFP-tree construction stage and parallel frequent pattern generation stage.

Parallel Projected XFP-tree Construction Stage

The process constructs a local projected XFP-tree from its data partition. In the first database scan, data is partitioned equally into $M \times N$ parts and each part is assigned to a thread where N is the number of processes and M is the number of threads per process. The process and its threads collaborate to compute the global count of all items by reducing the local count lists into a global one. They then identify the frequent items and sort them in frequency descending order. Figure 2 illustrates this compute task for the sample dataset in Table 1 with the execution model of 2 processes and 2 threads per process.

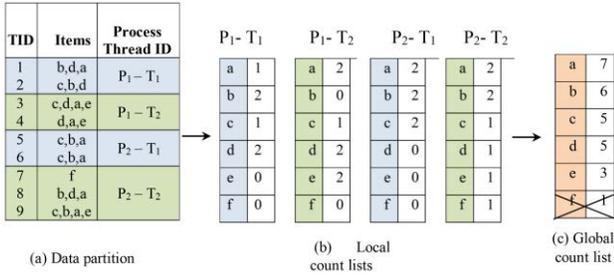


Figure 2: Computation of the global count by all processes (P = process; T = thread)

SDFEM then distributes the frequent items to processes in cyclic fashion [35]. For example, if we have two processes, P_1 and P_2 , and a list of frequent items a, b, c, d, e , then P_1 will mine all frequent patterns ending with item a, c, e and P_2 will mine all frequent patterns ending with b, d . In the second database scan, each process is responsible for the entire database; each thread reads a $1/M$ of database and filters transactions containing the assigned frequent items to construct a local FP-tree (Figure 3) and connect them into projected XFP-tree (Figure 4). This tree also ensures that each process can work independently. The cyclic scheduling balances the data size of each tree.

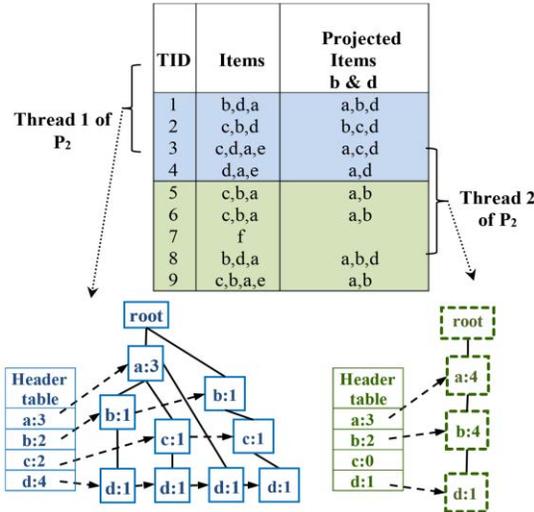


Figure 3: Construction of local FP-trees by each thread of P_2

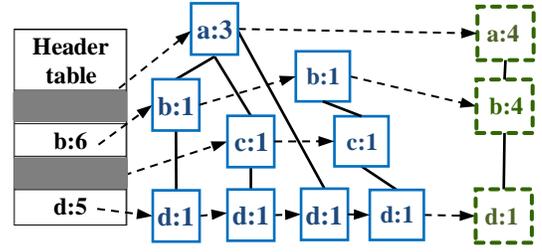


Figure 4: The project XFP-tree constructed by Process P_2

Parallel Frequent Pattern Generation Stage

After the first stage, each process has a projected XFP-tree in its memory. The process starts to independently generate frequent patterns using a multi-strategy mining approach inherited and improved from our prior sequential algorithms whose efficiency on both sparse and dense data has been shown in [29]. This new multi-strategy mining process uses ParallelMinePattern, MineFPtree, MineBitVector and LoadBalancing. The frequent pattern generation model for each thread of a process is illustrated in Figure 5.

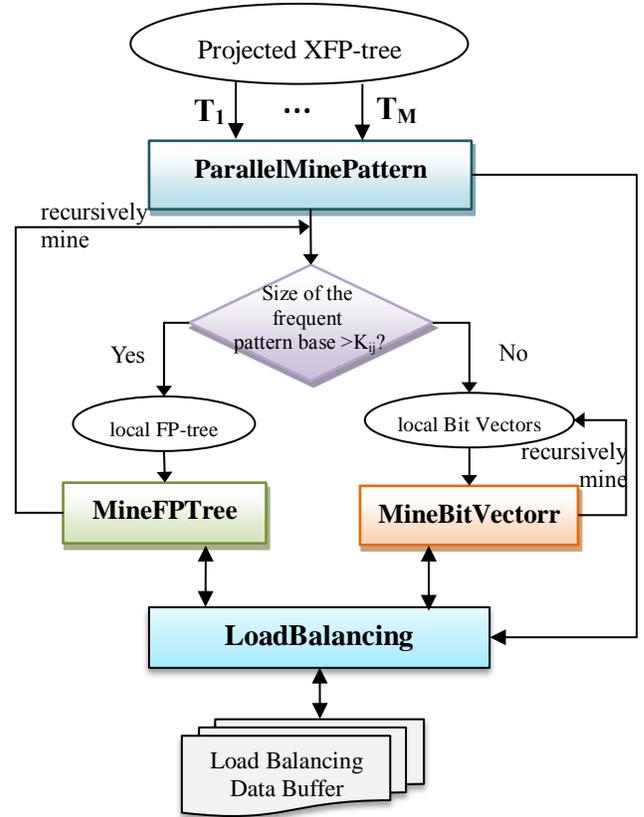


Figure 5: The mining model of SDFEM by a thread (T) within a process

ParallelMinePattern initializes the frequent pattern generation and manages the work distribution among parallel threads using dynamic job scheduling. Each thread uses this task to obtain a frequent item α in the header table of the shared projected XFP-tree. The thread invokes either *MineFPTree* or *MineBitVector* to generate the frequent patterns based on data characteristics [29].

MineFPTree is one of the two mining strategies in SDFEM. It generates frequent patterns by concatenating the suffix pattern of the previous step with each item α of the input FP-tree. Then, it constructs a child FP-tree for every item α using a data subset that is extracted from the input FP-tree and consists of sets of frequent items co-occurring with the suffix pattern. The new tree is then used as the input of this recursive mining task. This mining approach has been shown to perform well on sparse databases [36, 37, 38]. *MineFPTree* can switch to *MineBitVector* when it detects that the current data subset is dense by using a K_{ij} threshold values (one for each thread and it is computed using the method presented in [29]).

MineBitVector is the second mining strategy. It generates frequent patterns by concatenating the suffix pattern with each item of the input bit vector. It then joins pairs of bit vectors using a logical AND operation and computes their *support* using the weight vector to specify new frequent patterns. The resulting bit vectors are used as the input of *MineBitVector* to find longer frequent patterns. The mining process continues in a recursive manner until all frequent patterns are found. The efficiency of using the vertical data format on dense data has been shown in [8, 34, 39, 40]. *MineBitVector* is distinguished from the previous works because it uses a compact form of bit vectors where the compactness is presented in a weight vector.

LoadBalancing is another advanced feature of SDFEM for efficient deployment of FPM on a cluster environment because the workload associated with each item or itemset varies depending on the *minsup* input value. Hence, all threads use *LoadBalancing* to maintain workload balance during the mining process as described in more details in the next section.

MULTI-LEVEL LOAD BALANCING OF SDFEM

SDFEM is designed with two levels of parallelism: thread parallelism on shared memory multicores, and process parallelism on cluster nodes, which require minimizing communication for scalability. Therefore, load balancing in SDFEM also includes two levels: within-node load balancing for threads and between-node load balancing for processes. Balancing workload in a parallel task can be done implicitly with job scheduling and explicitly with load balancing techniques such as work sharing or work stealing. To maximize workload balance, we apply four load balancing techniques in SDFEM (Table 2). For simplicity, we implement a single data structure called load balancing

data buffer (LBDB) used by LoadBalancing for both within-node and between-node load balancing purposes.

Table 2: Load balancing techniques applied in SDFEM

	Within-node load balancing	Between-node load balancing
Implicit techniques	Dynamic Job Scheduling	Static Cyclic Scheduling
Explicit techniques	Work Sharing	Work Stealing

Within-node Load Balancing

SDFEM applies dynamic job scheduling and work sharing to maintain the workload balance and optimal CPU utilization among the threads of the same process.

Dynamic job scheduling: threads of the same process dynamically obtain the next available item from the header table of the projected XFP-tree and complete mining all frequent patterns ending with this item. In OpenMP, this is implemented by simply defining a dynamic directive for the parallel loop.

Work sharing: dynamic job scheduling is efficient. It however does not ensure load balance for cases where number of frequent items is considerably small, dense databases for example. A load balancer is added to each process; it maintains a load balancing data buffer holding shared data subsets. Busy threads within a process share their workload. Available threads seek additional work from this buffer. Because this data structure is shared among threads of the same process, all threads within a process can easily update it by using critical section or lock to ensure data integrity. Figure 6 illustrates the load balancing using work sharing.

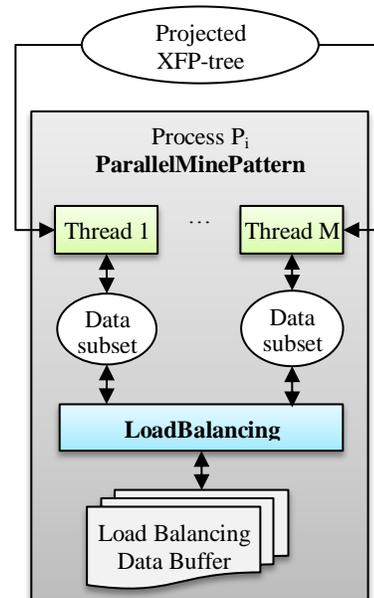


Figure 6: Within-node load balancing with work sharing

During the frequent pattern generation stage, threads periodically check this data buffer via *LoadBalancing*. If it is empty or if the number of data subsets is smaller than a certain limit Max_{LBDB} , threads will add their newly generated data subsets to the buffer, which are either FP-tree or Bit Vectors depending on the mining strategy being used (i.e. *MineFPTree* or *BitVector*). When a thread completes mining its data, it increments its counter TC by one where TC is a variable used to count number of threads completing mining assigned by the dynamic job scheduler). It then checks the data buffer to take a data subset and recursively mines this data subset. If the buffer is empty, the thread will wait until new data subsets are added, or until process status changes to *terminating*. This load balancing method ensures that threads of the same process remain busy until they all complete generating frequent patterns for the process's data partition. To minimize memory usage and the overhead of maintaining data buffer, we keep the number of buffer entries as small as possible. For example, in our experiments, the maximum number of buffer entries was set to the total number of threads of all processes.

Between-node Load Balancing

SDFEM applies static cyclic job scheduling and work stealing techniques to balance workload among the processes.

Cyclic job scheduling: due to the large amount of processed data, dynamic job scheduling may result in huge communication overhead and bottleneck. We apply static job scheduling to distribute work among processes because it is simple, has practically no overhead and does not require communication and synchronization optimization. In the first stage of projected XFP-tree construction after the frequent items are found and sorted in frequency descending order, they are distributed to processes in a cyclic fashion as illustrated in Stage 1. Each process filters database using the assigned items to construct XFP-tree.

Work stealing: in most cases when the number of frequent items are large, cyclic job scheduling is good for initial work partition among parallel processes. We apply work stealing to maintain better load balance, especially for cases where mining workload is associated with significantly varied frequent items or the number of frequent patterns is small. Based on work stealing techniques, idle processes actively look for busy processes to request more work. Since only idle processes attempt to communicate, the amount of communication is reduced and the overhead is well tolerated compared to idle time of processes without work [21]. Both work sharing and work stealing use the same data buffer. On starting the frequent pattern generation, each process keeps a process status list whose elements indicate the status of all processes (i.e. *working*: a process is still generating frequent patterns from its pre-scheduled data, *balancing*: a process completed its work

portion and is requesting more work from a remote process, *terminating*: a process completed its work and there is no *working* process to request for more work). A process's status is initialized as *working*. If a process P_k completes generating all frequent patterns from its projected XFP-tree (i.e. a *balancing* process) it will pick up a victim process P_h among the *working* processes and sends a job request to P_h . Figure 7 depicts the load balancing model with work stealing between P_k and P_h .

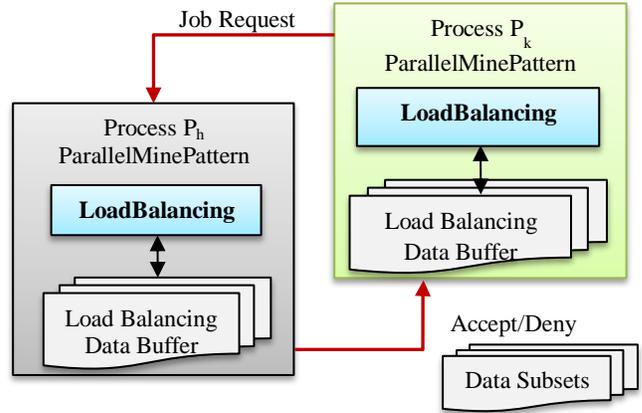


Figure 7: Between-node load balancing with work stealing. Although work stealing technique requires communication among processes, its overhead is always small because of the following reasons. First, SDFEM employs the hybrid programming model that create fewer parallel processes (i.e. usually equal to the number of nodes in the cluster). Because the number of processes involved in load balancing is small and data communication is decentralized due to work stealing, the overhead of load balancing is much smaller than that of parallel programming with pure message passing. Second, in SDFEM, only one thread, the master thread, of each process participates in work stealing while the other threads continue its mining work.

PERFORMANCE EVALUATION

Experimental Setup

Datasets: The five real datasets used for our experiments: two sparse, one moderate and two dense databases are obtained from the FIMI Repository [41], a well-known repository for FPM. The database features are reported in Table 3.

Table 3: Experimental datasets of SDFEM

Dataset	Type	# of Items	Average Length	# of Trans.
Chess	Dense	76	37	3196
Pumsb	Dense	2113	74	49046
Accidents	Moderate	468	33.8	340183
Kosarak	Sparse	41271	8.1	990002
Webdocs	Sparse	52676657	177.2	1623346

Software: In our experiments, we use OpenMP and OpenMPI to implement SDFEM; g++ (OpenMP) and mpic++ (OpenMPI) for compilation.

Hardware: We use the cluster at <http://pds.ucdenver.edu> consisting of several Altus 1702 machines where each node is equipped with dual AMD Opteron 2427 processor, 2.2GHz, 24GB memory and 160 GB hard drive. The interconnection among nodes is Infiniband. Benchmark experiments of SDFEM with up to 120 cores (i.e. 10 nodes of our cluster) were conducted. The operating system is CentOS 5.3, a Linux-based distribution.

Execution Time

We demonstrate the performance of SDFEM by measuring its execution time for various number of cores of the test cluster on five datasets. The sequential test mode was done on 1 core. The parallel test was done by running the program on varying number of nodes from 1 to 10, each node runs up to 12 cores, providing a range of 12 to 120 threads or cores. Experimental results of SDFEM with varying number of cores are shown in Figure 8.

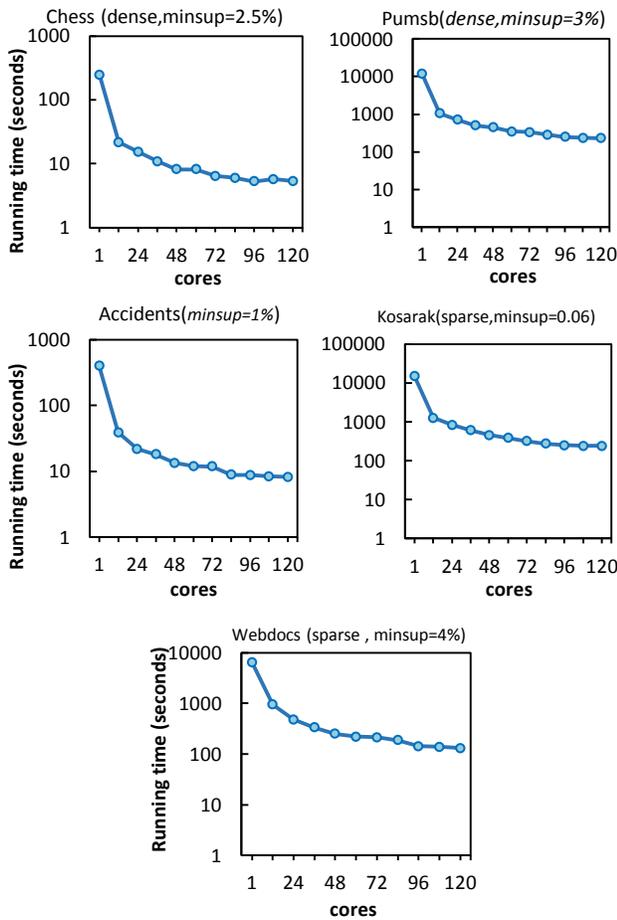


Figure 8: Running time of SDFEM (from 1 to 120 cores)

The results show significant reduction in execution time is obtained for all test cases. In the experiments, SDFEM

reduces the mining time on Webdocs databases from 6460 seconds on 1 core to 130 seconds on 120 cores which saves 97.98% of the time required by sequential execution (i.e. $97.98\% = (6460-130)/6460 * 100\%$). The time of SDFEM on Kosarak database is cut down 98.4% (from 15234 seconds on 1 core to 238 seconds on 120 cores). Similarly, the percentage of execution time savings for Chess, Pumsb and Accidents are 97.79%, 98.08%, 97.95% respectively. This performance improvement comes from sharing mining workload for large number of cores and reducing the amount of data that each parallel process/thread has to handle. Performance gains of SDFEM is consistent for both dense and sparse databases.

Speedup

To evaluate scalability of SDFEM to the size of cluster, we compute its speed up by dividing the sequential time of SDFEM by the parallel execution time (i.e. for 12, 24, 36, ... 120 cores) and present the results in Figure 9.

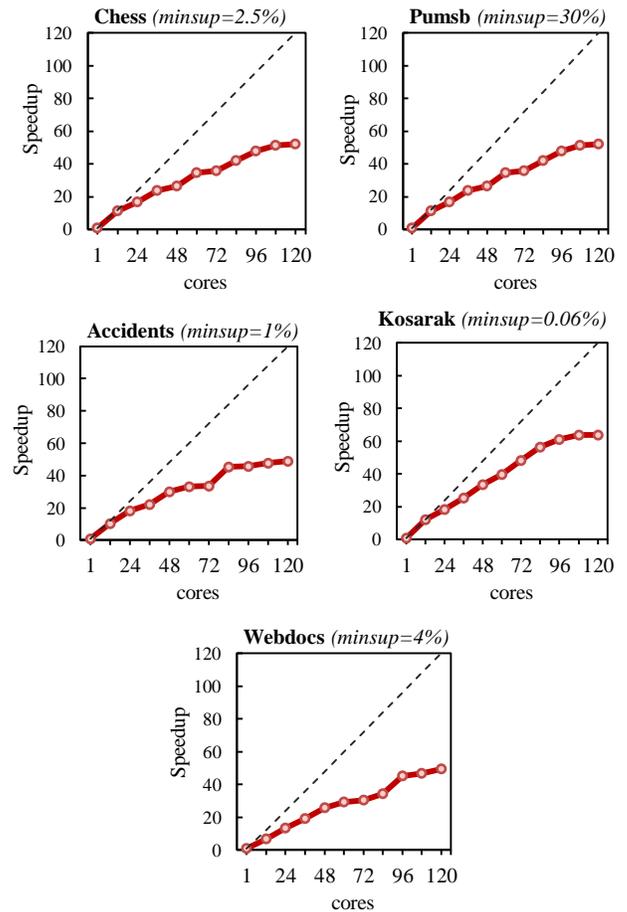


Figure 9: Speedup of SDFEM (from 1 to 120 cores)

We can see that for most cases, speed up increases when the number of cores is increased. Speed up values of five datasets on 120 cores are 45.4 (Chess), 52.1 (Pumsb), 64.8

(Kosarak), 48.7 (Accidents), 49.6 (Webdocs). Many factors limit scalability of most parallel FPM methods like synchronization, load balancing and data communication overheads or limitations of test hardware like serial I/O. It is important to note that, as the number of nodes (multiples of 12 in plots of Figure 9) is increased, higher speedups are obtained which shows SDFEM scales well for larger machines.

Impact of Hybrid MPI-OpenMP Programming Model

Application of hybrid MPI-OpenMP programming model is an important feature of SDFEM, distinguishing it from related work. We study the impact of this programming model in comparison with the traditional pure MPI model. For this purpose, SDFEM has been benchmarked using 5 compute nodes with 60 cores in total in two scenarios:

Scenario 1- 1 process per core and total processes = 60

Scenario 2- 12 threads per process, 1 process per node and total threads = 60

In Scenario 1, SDFEM performs exactly like a traditional MPI program. In Scenario 2, SDFEM applies the hybrid programming model presented in Section 3. The comparison results in Table 4 shows clear evidence that using hybrid model can significantly improve the performance. Compared to pure MPI, the hybrid mode with 12 threads per process enhances from 78.6% up to 329.5% of mining performance. For Kosarak, SDFEM with the hybrid model runs much faster than its pure MPI version (474 seconds vs. 2036 seconds).

Table 4: Time comparison of pure MPI vs. hybrid MPI-OpenMP (60 cores)

Datasets	MinSup	Scenario 1 Pure MPI (1) (sec.)	Scenario 2 Hybrid (2) (sec.)	Performance Improvement (4)= $((2)-(1))*100/(2)$
Chess	2.50%	42	11	281.8%
Pumsb	30%	128	68	88.2%
Accidents	1%	25	14	78.6%
Kosarak	0.06%	2036	474	329.5%
Webdocs	4%	436	220	98.2%

Impact of Different Load Balancing Techniques

Another important factor impacting FPM performance is load balancing. We evaluate the efficiency of the four load balancing techniques applied in SDFEM by implementing 4 different versions of SDFEM where each integrates a combination of different techniques as listed in Table 5 and benchmarking them using 120 cores (12 threads/cores per process, 1 process per nodes). In Table 5, the underlined values indicate load-balancing techniques applied in SDFEM. All load balancing techniques are applied in

SDFEM-V4 while fewer are used in the others: SDFEM-V1 (1 technique), SDFEM-V2 (2 techniques) and SDFEM-V3 (3 techniques). The test results presented in Table 6 show that SDFEM-V4 runs much faster the other three versions, showing clearly the importance of load balancing techniques. For example, for Kosarak dataset, SDFEM_V4 runs 8.7 times faster than SDFEM_V1 ($8.7=2153/248$), 8.1 times faster than SDFEM_V2 ($8.1=2020/248$), and 1.2 times faster than SDFEM_V3 ($1.2=304/248$).

Table 5: Four versions of SDFEM with different load balancing techniques

Techniques	SDFEM-V1	SDFEM-V2	SDFEM-V3	SDFEM-V4
Within-node job scheduling	Static (cyclic)	<u>Dynamic</u>	<u>Dynamic</u>	<u>Dynamic</u>
Work Sharing	N/A	N/A	<u>Yes</u>	<u>Yes</u>
Between-node job scheduling	<u>Static (cyclic)</u>	<u>Static (cyclic)</u>	<u>Static (cyclic)</u>	<u>Static (cyclic)</u>
Work Stealing	N/A	N/A	N/A	<u>Yes</u>

Table 6: Running time of four versions of SDFEM

Datasets	MinSup	SDFEM-V1 (sec.)	SDFEM-V2 (sec.)	SDFEM-V3 (sec.)	SDFEM-V4 (sec.)
Chess	2.5 %	42.2	42.3	7.5	5.2
Pumsb	30 %	3153	3152	341	252
Accidents	1 %	44	27	9.7	8.1
Kosarak	0.06 %	2153	2020	304	248
Webdocs	4 %	637	316	130	129

CONCLUSION

We present SDFEM, a novel parallel FPM for multi-core clusters, as a high performance FPM solution for large-scale applications. SDFEM has three main features which have not been investigated by prior parallel FPM work. They include (1) use of hybrid programming model to leverage benefits of shared memory and enhance the mining performance; (2) application of multiple load balancing techniques to achieve high performance and scalability; and (3) utilization of data characteristics-based mining approach that we developed to perform efficiently on different types of data. Our performance evaluation has shown that in our test cases, SDFEM results in savings of 97.79% - 98.4 % compared to sequential time. SDFEM on 120 cores of our cluster runs 45.4 – 64.8 times faster than its sequential version for the test datasets. The execution time of SDFEM are over the complete program execution and includes I/O time and the cost of performing multiple reduction steps and balancing workload.

REFERENCES

1. Agrawal, R. and Srikant, R. Fast Algorithms For Mining Association Rules In Large Databases. In *Proc. 20th International Conference on Very Large Data Bases* (1994), 487-499.
2. Brin, S., Motwani, R. and Silverstein, C. Beyond Market Baskets: Generalizing Association Rules To Correlations. In *Proc. the 1997 ACM SIGMOD international conference on Management of data* (1997), 265-276.
3. Silverstein, C., Brin, S., Motwani, R. and Ullman, J. Scalable Techniques for Mining Causal Structures. *Data Mining and Knowledge Discovery* (2000), vol. 4, no. 2-3, 163-192.
4. Agrawal, R. and Srikant, R. Mining Sequential Patterns. In *Proc. the Eleventh International Conference on Data Engineering* (1995), 3-14.
5. Mannila, H., Toivonen, H. and Verkamo, A. Inkeri. Discovery of Frequent Episodes in Event Sequences. *Data Mining and Knowledge Discovery* (1997), vol. 1, no. 3, 259-289.
6. Han, J., Dong, G. and Yin, Y. Efficient Mining of Partial Periodic Patterns in Time Series Database. In *Proc. the 15th International Conference on Data Engineering* (1999), 106-115.
7. Han, J., Cheng, H., Xin, D. and Yan, X. Frequent Pattern Mining: Current Status And Future Directions. *Data Mining and Knowledge Discovery* (2007), vol. 15, no. 1, 55-86.
8. Burdick, D., Calimlim, M., Flannick, J., Gehrke, J. and Yiu, T. MAFIA: A Maximal Frequent Itemset Algorithm. *IEEE Transactions on Knowledge and Data Engineering* (2005), vol. 17, no. 11, 1490-1504.
9. Chorley, M. J. and Walker, D. W. Performance Analysis of a Hybrid MPI/OpenMP Application on Multi-core Clusters. *Journal of Computational Science* (2010), vol. 1, no. 3, 168-174.
10. Rabenseifner, R., Hager, G. and Jost, G. Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. in *Proc. the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing* (2009), 427-436.
11. He, Y. and Ding, C. H. MPI and OpenMP Paradigms on Cluster of SMP Architectures: the Vacancy Tracking Algorithm for Multi-Dimensional Array Transposition. *Supercomputing, ACM/IEEE 2002 Conference* (2002), 1-14.
12. Wu, X. and Taylor, V. Performance Characteristics of Hybrid MPI/OpenMP Implementations of NAS Parallel Benchmarks SP and BT on Large-scale Multicore Supercomputers. *SIGMETRICS Perform. Eval. Rev.* (2011), vol. 38, no. 4, 56-62.
13. Hager, G., Jost, G. and Rabenseifner, R. Communication Characteristics And Hybrid MPI/OpenMP Parallel Programming On Clusters Of Multi-Core SMP Nodes. in *Proc. Cray User Group Conference* (2009), p.5455.
14. Liu, L., Li, E., Zhang, Y. and Tang, Z. Optimization of Frequent Itemset Mining on Multiple-Core Processor. in *Proc. the 33rd international conference on Very large data bases* (2007), 1275-1285.
15. Zaki, M. Parallel and Distributed Association Mining: A Survey. *IEEE Concurrency* (1999), vol. 7, no. 4, 14-25.
16. Garg, R. and Mishra, P. K. Some Observations of Sequential, Parallel and Distributed Association Rule Mining Algorithms. in *Proc. the 2009 International Conference on Computer and Automation Engineering* (2009), 336-342.
17. Moonesinghe, H. D. K., Chung, M. and Tan, P. Fast Parallel Mining of Frequent Itemsets. in *Michigan State University*.
18. Tanbeer, S. K., Ahmed, C. F. and Jeong, B.-S. Parallel and Distributed Frequent Pattern Mining in Large Databases. in *Proc. the 2009 11th IEEE International Conference on High Performance Computing and Communications* (2009), 407-414.
19. Yu, K.-M., Zhou, J., Hong, T.-P., and Zhou, J.-L. A Load-Balanced Distributed Parallel Mining Algorithm. *Expert Systems with Applications* (2010), vol. 37, no. 3, 2459-2464.
20. El-hajj, M. and Zaïane, O. R. Parallel Leap: Large-scale Maximal Pattern Mining In A Distributed Environment. in *Proc. 12th International Conference on Parallel and Distributed Systems* (2006).
21. Dinan, J., Olivier, S., Sabin, Prins, G. J., Sadayappan, P., and Tseng, C.-W. Dynamic Load Balancing of Unbalanced Computations Using Message Passing. *Parallel and Distributed Processing Symposium, International* (2007), vol. 0, p. 391.
22. Yu, K.-M., Zhou, J. and Hsiao, W. Load Balancing Approach Parallel Algorithm for Frequent Pattern Mining. *Parallel Computing Technologies* (2007), vol. 4671, V. Malyshkin, Ed., Springer Berlin Heidelberg, 623-631.
23. Manaskasemsak, B., Benjamas, N., Rungsawang, A., Surarerk, A. and Uthayopas, P. Parallel Association Rule Mining Based On FI-Growth Algorithm. in *Proc. 2007 International Conference on Parallel and Distributed Systems* (2007), 1-8.
24. Ramaiah, B. Janaki, Reddy, A. Rama Mohan, and

- Kumari, M. Kamala. Parallel Privacy Preserving Association Rule Mining on PC Clusters. *IEEE International Advance Computing Conference* (2009), 1538-1542.
25. Yu, K.-M., and Zhou, J. Parallel TID-based Frequent Pattern Mining Algorithm on a PC Cluster and Grid Computing System. *Expert Syst. Appl.* (2010), vol. 37, no. 3, 2486-2494.
 26. Tseng, F. S., Kuo, Y.-H., and Huang, Y.-M. Toward Boosting Distributed Association Rule Mining By Data De-clustering. *Information Sciences* (2010), vol. 180, no. 22, 4263-4289.
 27. Ozkural, E., Ucar, B., and Aykanat, C. Parallel Frequent Item Set Mining with Selective Item Replication. *IEEE Transactions on Parallel and Distributed Systems* (2011), vol. 22, no. 10, 1632-1640.
 28. Vu, L. and Alagband, G. A Fast Algorithm Combining FP-Tree and TID-List for Frequent Pattern Mining. In *Proc. the 2011 International Conference on Information and Knowledge Engineering* (2011).
 29. Vu, L. and Alagband, G. Mining Frequent Patterns Based on Data Characteristics. In *Proc. the 2012 International Conference on Information and Knowledge Engineering* (2012).
 30. Vu, L. and Alagband, G. An Efficient Approach for Mining Association Rules from Sparse and Dense Databases. In *Proc. the 2014 International Conference on Information and Knowledge Management, IEEE,* (2014).
 31. Vu, L. and Alagband, G. Efficient Algorithms for Mining Frequent Patterns from Sparse and Dense Databases. *Intelligent Systems* (2014).
 32. Pramudiono, I., and Kitsuregawa, M. Parallel FP-growth on PC Cluster. in *Proc. the 7th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining* (2003).
 33. Sucahyo, Y., Gopalan, R. and Rudra, A. Efficiently Mining Frequent Patterns from Dense Datasets Using a Cluster of Computers. in *AI 2003: Advances in Artificial Intelligence*, vol. 2903, T. Gedeon and L. Fung, Eds., Springer Berlin Heidelberg, (2003), 233-244.
 34. Zaki, M., Parthasarathy, S., Ogihara, M. and Li, W. New Algorithms for Fast Discovery of Association Rules. in *Proc. the 3rd International conference on Knowledge Discovery and Data Mining* (1997).
 35. Jordan, L. E. and Alagband, G. *Fundamentals of Parallel Processing*, Prentice Hall Professional Technical Reference (2002).
 36. Han, J. Pei, J. and Yin, Y. Mining Frequent Patterns Without Candidate Generation. in *Proc. the 2000 ACM SIGMOD international conference on Management of data* (2000).
 37. Grahne, G. and Zhu, J. Efficiently Using Prefix-trees in Mining Frequent Itemsets. In *Proc. the 2003 Workshop on Frequent Pattern Mining Implementations* (2003).
 38. Racz, B. nonordfp: An FP-Growth Variation without Rebuilding the FP-Tree. In *Proc. the 2004 Workshop on Frequent Pattern Mining Implementations* (2004).
 39. Shporer, S. AIM2: Improved Implementation of AIM," in *Proc. the 2004 Workshop on Frequent Itemset Mining Implementations* (2004).
 40. Schmidt-Thieme, L. Algorithmic Features of Eclat. In *Proceedings of the 2004 Workshop on Frequent Itemset Mining Implementations* (2004).
 41. Frequent Itemset Mining Implementations Repository. In *Workshop on Frequent Itemset Mining Implementation, 2003-2004*.

Efficient Scaling of a Hydrodynamics Simulation Using Compiler-based Accelerator Technology

Jordan Bradshaw

University of South Carolina
bradshja@email.sc.edu

Phil Moore

University of South Carolina
phil@sc.edu

Behzad Torkian*

University of South Carolina
torkian@sc.edu

ABSTRACT

In the realm of numerical modeling, there is often a requirement to run simulations with higher spatial and temporal resolutions. Increasing resolution can improve the accuracy of simulations with a corresponding increase in the run time. These run times can become impractical for conventional sequentially coded simulations.

Parallelizing simulation code offers great possibilities for improved run-time performance, but it can be very difficult to achieve the necessary speedups when software development and debugging time are considered. This is especially the case when extensive modifications to legacy source code are needed to incorporate parallel processing capabilities.

To address this problem, we used a directive-driven compiler to parallelize a hydrodynamics simulation targeting GPU hardware. This compiler-level parallel development environment enabled us to parallelize an existing legacy program with minimal alterations to the original source code.

Using compiler directives and testing methodologies, we were able to achieve more than a 15x speedup and reduce the run time of high resolution simulations from months to days while verifying that results were equivalent to the sequential legacy version of the simulation.

Author Keywords

Simulations; GPU; Compilers; Accelerators; OpenACC; CUDA; Hydrodynamics

ACM Classification Keywords

C.4 PERFORMANCE OF SYSTEMS; D.1.3
CONCURRENT PROGRAMMING

INTRODUCTION

Simulations from a vast breadth of scientific fields have come to exploit and rely on the continued increase in computational capabilities provided by modern computers. As advances in technology provide more computational power, computer simulations are generating datasets with greater resolutions to take advantage of this processing ability.

The trend during the past decade has focused more on providing additional processor cores rather than a direct increase in processing speed. As a result, programmers face considerable difficulties in accelerating legacy simulation code on the new architectures. Low-level parallel programming is an inherently time-consuming task which is made even more complicated by the need to integrate parallel libraries with long-established applications developed to run on a single processor.

OpenMP [4] is a commonly used API for parallelizing existing applications with minimal changes, but is targeted at conventional multiprocessor systems that have limited scaling potential.

More exotic parallel architectures, such as GPUs, can offer greater speedup potential, but require that applications be rewritten to use new and sometimes proprietary libraries. Nvidia's CUDA [1] and the similar non-proprietary OpenCL [2] languages are examples of GPU specific application programming interfaces that require computational kernels be rewritten in a specialized syntax which is often a large and error prone undertaking.

A more recent development to help ease the difficulty of working with hardware 'accelerators' is the OpenACC [3] specification which seeks to provide a standardized interface for compilers and accelerator hardware such as GPUs. OpenACC is a directive-based compiler-driven set of language extensions that allows programmers to ignore most of the underlying details of how the targeted

accelerator works and instead focus on program correctness and tweaking compiler directive options for performance.

In this paper, we provide an example of using OpenACC directives to achieve a significant speedup of a legacy hydrodynamics simulation with a focus on minimizing alterations to the code and maintaining accuracy of the simulation's results. Specifically, we make the following contributions:

- We describe the hydrodynamics simulation ALGE3D [6] used by the Department of Energy and demonstrate problems in scaling when faced with significant increases in resolution.
- We describe the OpenACC specification, its implementation in the Portland Group (PGI) [5] Fortran Accelerator compiler and how it allows legacy Fortran programs to be offloaded to GPU accelerators.
- We provide a methodology that we developed to iteratively offload segments of ALGE3D to the GPU which allowed us to maintain consistent results while improving run-time performance.

This paper is organized as follows: we give a summary of prior work related to parallelizing hydrodynamics simulations, introduce ALGE3D including its purpose and design, introduce PGI's CUDA Accelerator, examine our strategy for parallelizing code with the accelerator then show the results of this work.

PRIOR WORK

The use of compiler driven program parallelization for hydrodynamics simulations has been investigated extensively in the past using standards such as OpenMP [7] [8][10] and MPI [8][10] which provided modest speedups of less than 10x.

Dong *et al.* [9] demonstrated that hand-coded CUDA kernels can achieve excellent speedups on Nvidia GPUs especially when combined with OpenMP on host CPUs, but their work is done without automatic parallelization aids. Similarly, Schive *et al.* [11] and Bernaschi *et al.* [12] show very large performance gains with hand written CUDA kernels.

Some work has already been done to show that automatic translation of parallel code into GPU kernels can be effective as described by Lee and Eigenmann's OpenMPC [18]. OpenMPC is an extension of the OpenMP API that works as a translation and tuning framework for GPUs. They were able to achieve speedups approaching 10x for some scientific applications.

In the case of PGI's CUDA Accelerator, Wienke *et al.* [13] have shown that PGI's compiler is able to produce kernels that compete with hand written OpenCL kernels and are faster than OpenACC kernels generated by the Cray

compilers. They do not show how these results compare to baseline CPU implementations.

Herdman *et al.* [14] demonstrated that OpenACC-based compiler directives can produce efficient code that competes with hand written CUDA and OpenCL kernels. Their results focus more on programmer productivity and are specific to the Cray architecture.

ALGE3D

The simulation that we chose to accelerate is ALGE3D – a Fortran hydrodynamics program developed by the Department of Energy at the Savannah River National Laboratory to perform simulations of heat flow and dissipation in cooling ponds then later extended to include tracer flow simulations in environments that experience periodic wetting and drying. ALGE3D has been used by the DOE since 1997 to perform simulations that estimate the behavior of spills and thermal plumes [19][20][21].

Algorithmic Details

At its core, ALGE3D solves a series of mathematical equations that account for the kinetic energy, thermal energy and particulate density of water in a grid fashion. Environmental factors such as solar radiation, atmospheric interactions, cloud cover and tidal flows leading to wetting and drying all factor into these equations. The body of water being studied is divided into a three-dimensional grid of uniform resolution with the equations being solved for each cell in a variable time step locked fashion. Each quantity being simulated is stored separately as different grids in memory.

While most of the components of the simulation depend on neighboring cells when performing calculations, the time step nature allows the next timestep's grids to be populated from current values and allow each cell to be calculated independently in parallel for many parts of the simulation.

Performance Characteristics

ALGE3D's performance is closely tied to the spatial resolution of the simulation being run. Increasing the spatial resolution leads to a corresponding increase in the number of grid cells that must be solved per time step and thus runtime.

At the onset of the project, it was desired to run a simulation at a minimum resolution of 1 m³ on a total volume of 16,998,668 m³ for a total of 335 days of simulation time. Projections indicated that using a single core of an Intel i7-4770K at 3.5 Ghz with 4 GB of dual channel DDR3 RAM would take over two months to complete a simulation at this resolution. This was an unacceptable turnaround time for obtaining results. A speedup of greater than 10x would bring it back in line with the speed of the previous simulations that completed in a week or less.

Acceleration Potential

Gaining such a significant speedup would require parallelizing ALGE3D in some manner. Traditional multiprocessor options using OpenMP and MPI were considered as was the more recent GPU architecture and associated libraries.

MPI was ruled out because we wanted to parallelize ALGE3D by making as few changes as possible to its source code. MPI would have required substantial modifications to the source code for an unknown potential speedup. OpenMP would have been more feasible, but the potential speedup likely would have been fairly modest in even the best case.

GPUs provide a comparatively large speedup. Although ALGE3D's algorithms are highly parallel with each grid cell able to be evaluated separately in many computation kernels, converting existing code to run on a GPU is an involved task. ALGE3D consists of over 5,000 lines of Fortran code. Porting its kernels to run on a GPU would be infeasible due to the extensive source code modifications required and the need to rewrite extensive sections in a language with native CUDA bindings such as C.

Nevertheless, the speedup potential of a GPU led us to seek an alternate way to use ALGE3D on a GPU. The solution was to use a directive-driven compiler accelerator developed by PGI that includes OpenACC directives to convert existing Fortran code to equivalent CUDA code for execution on a GPU.

PGI ACCELERATOR AND OPENACC

To facilitate compiler-driven code parallelization, PGI's CUDA Accelerator, based on the OpenACC standard, was chosen. It is a vendor-specific implementation that targets Nvidia CUDA-enabled GPUs.

OpenACC

OpenACC is intended to be a standard for code to utilize hardware accelerator technologies including but not limited to GPUs. It relieves the programmer of burdens related to initializing or shutting down the target accelerator, generating or transferring code to it and handling data movement between host and accelerator. We will show later that the automatic data management is generally suboptimal and programmers will see much greater benefits by manually controlling data movement.

```
!$acc region
do j=1,ny
  do i=1,nx
    foo(i, j) = 2 * bar(i, j)
  end do
end do
!$acc end region
```

Figure 1: OpenACC Directives for a do-loop

OpenACC takes the form of compiler directives, implemented as specially formatted code comments, that either directly instructs the compiler to perform some action such as performing a data copy to the accelerator or serves as a means of marking blocks of code for parallelization. An example of a Fortran do-loop marked for parallelization is shown in Figure 1.

PGI Accelerator

PGI's implementation of the OpenACC standard, which they term PGI Accelerator, is specifically targeted at converting C, C++ and Fortran code for execution on CUDA-enabled GPUs.

In addition to providing directives for marking parallel loops, PGI Accelerator also supports directives for manual memory management on the accelerator, data movement, code offloading, automatic loop parallelization and manual loop scheduling.

PGI Accelerator is an extension to the PGI C, C++ and Fortran compilers and is enabled by a command line option which allows forward compatibility by specifying a targeted accelerator architecture. For our experiments, we were building for the NVIDIA Tesla architecture only.

PARALLELIZATION METHODOLOGY

While achieving a significant speedup was the goal, it was crucial that the results of sequential code running on a traditional CPU and parallel code running on a GPU were the same. We discovered that there were a number of potential pitfalls with using the accelerator and GPU regarding the simulation's accuracy. To minimize the difficulty of finding and dealing with these sources of error, we developed and adhered to a strict parallelization strategy. Below, we detail some of these sources of error and then describe our strategy for minimizing them.

Sources of Error

Floating Point Math

Hardware used to perform fast floating point math on modern computers has a fixed bit length which places a limit on the amount of precision that can be achieved when performing such calculations.

While all CUDA enabled GPUs implement the same IEEE 754 floating point standard [15] as the x86 family of CPUs used to generate baseline simulation data, internal differences exist that can cause discrepancies when using high precision calculations. Whitehead and Fit-Florea [16] give an excellent and in-depth overview of some of the differences which are described next.

Complications stem from the fact that some CPUs, such as Intel's x86 family processors, include extended internal space for calculations in their FPU. This means that while results are always presented in standard 32 or 64-bit floating point format, calculations that are done entirely in the FPU can benefit from extended precision up to 80 bits.

Since this extended precision is not available on CUDA-enabled GPUs, the difference in precision can result in small errors that may become apparent at very high resolutions or when simulating at very small timesteps where changes accumulate and are lost in standard precisions.

In our experience, this did not contribute measurably to ALGE3D's output since all of its calculations are done with double precision floating point math. GPU kernels operating in single precision, which is substantially faster on older CUDA-capable GPUs, may be susceptible.

Race Conditions

Parallel code has inherit time synchronization considerations that sequential code does not. Operations that are always executed in order in sequential code can't, in general, be expected to execute in that same order when distributed across multiple threads.

This requires that programmers rely on synchronization operations and primitives to enforce some level of order on the operations.

PGI's Accelerator automatically detects locations where synchronization is necessary and inserts the needed code to enforce consistent behavior. Programmers are also given access to these primitives in order to enforce sequential behavior in places that the compiler does not recognize as needing them.

Furthermore, the compiler may mistakenly believe that a section of code is not parallelizable but still provides directives to allow programmers to force generation of parallel code. This can introduce race conditions or other inconsistent or nondeterministic behavior if the programmer forces the compiler to parallelize code that truly is not independent across threads.

Compiler Bugs

Like all software of any complexity, compilers are not without bugs. We encountered bugs using PGI's compiler on several occasions. These bugs generally manifested as incorrectly generated CUDA code which could either produce erroneous results or cause the program to crash when run on the GPU.

As an example, we encountered two notable bugs related to the automatic management of shared memory. The first manifested whenever the compiler was instructed to parallelize a loop which operated backward, that is, by decrementing its induction variable rather than incrementing it. When this compiler directive was used, it would miscalculate the bounding indexes for any arrays used in the kernel then either copy too little or too much between the host and GPU causing garbage values to be copied.

Similarly, unusual computed array strides could cause the compiler to produce kernels that misused the GPU's shared memory. Shared memory is a cache memory local to a

group of threads and allows much faster access to values that are used in multiple adjacent threads. The only solution in this case was to disable automatic shared memory management altogether for the affected kernel.

To help diagnose these bugs, the compiler provides a command line switch to dump the automatically generated CUDA code to a file for viewing. However, this is only useful as a diagnostic for small kernels due to the machine-formatted nature of the generated code. Also, this CUDA code cannot be used as a base to modify since the compiler does not accept handwritten CUDA code.

Iterative Parallelization Workflow

The workflow for parallelization using PGI's CUDA Accelerator can be broken down into a series of simple steps:

Identify Parallel Code

GPUs are designed to operate in a massively parallel manner and are most efficient when a large number of threads can operate concurrently using the same code path.

To generate code that matches this multi-threading capability as closely as possible, PGI's accelerator targets loops in the sequential program. Any loop body that contains code which does not depend on other iterations of the loop is potentially parallelizable if it has enough iterations to be a suitable candidate for conversion using the compiler. Memory copies and launching kernels incur a large overhead. Small loops or those with few loops which could be translated as GPU threads will likely not benefit.

Other factors can influence whether a loop is well suited to parallelization include the memory access patterns, the length of the code block, control flow and subroutine calls. Modern GPUs suffer severe performance degradation when concurrent threads take different execution paths because the SIMD architecture forces all concurrent threads to execute the same instruction necessitating repeating the code block along the second path (referred to as a branch divergence). For the compiler that we used, beta version 12.3, subroutine calls can be problematic. The accelerator compiler suite required us to manually inline them which often required rewriting blocks of code to compensate.

While it is useful to consider manually identifying loops to parallelize, profiling, as detailed below, is the best way to determine whether parallelizing a particular loop is beneficial.

After identifying loops that are suitable candidates, the programmer adds accelerator directives around the code blocks to instruct the compiler to parallelize them. Figure 1 gives a simple example of the needed directives.

```

free_surface:
  21, Generating copyin(u(1:i), v(1:j))
  21, Generating copyout(u(1:i), v(1:j))
  22, Loop is parallelizable
  23, Loop is parallelizable
  Accelerator kernel generated
  21, !$acc region
    CC 2.0: 19 registers; 48 shared, 16 constant,
    0 local memory bytes; 100% occupancy

```

Figure 2: Sample Compiler Feedback

Parallelize

During the code analysis step, the compiler scans any code blocks delimited by !\$acc directives and attempts to determine if the code can be safely parallelized, and if so, what strategy to use.

To assist in parallelization, the compiler provides feedback to the developer identifying parallel code blocks, pointing out code that it believes to be unsafe to parallelize, determining the minimum memory arrays and values that need to be copied into or out of the kernel and the thread schedule used for the kernel. Figure 2 gives an example of the compiler's feedback on a parallelizable loop showing the memory ranges that need to be copied in and out as well as a summary of the GPU's memory usage and thread occupancy for the kernel.

Code is deemed unsafe to parallelize if any iteration of the loop depends on other iterations of the loop since each loop iteration is a separate thread on the GPU introducing race conditions. While the compiler is thorough and checks computed index references, it is occasionally overly conservative and may determine that loops that are independent are not safe to parallelize. To address this, the developer can force it to parallelize the code anyway using the *independent* clause in the loop schedule.

Ignoring the compiler's warnings can produce errors in the results. It is crucial that any parallelized code be verified by comparing its results with the sequential code's results.

Verify

Before significant effort is put into tuning the code for speed, results must be verified so that any errors or inconsistencies can be identified and corrected first.

Because the primary goal of parallelizing code is to reduce runtime requirements, running complete production simulations may be infeasible or too time consuming especially when working iteratively. For this reason, small or short simulations can be useful to project the behavior for larger or more comprehensive simulations when verifying equivalent results from the GPU and sequential code.

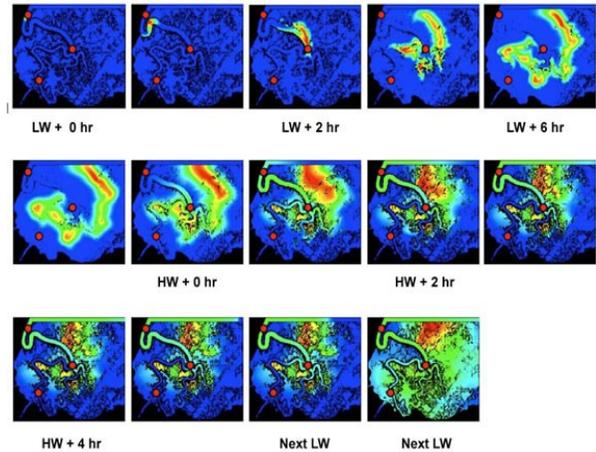


Figure 3: Dye Tracer Output in Tecplot

In ALGE3D, a set of grids are output that represent values such as water levels, temperature, and tracer concentrations. This makes verification straightforward. The resulting numbers can be compared directly against the reference sequential CPU version. For fast visual verification, we also used graphical software TecPlot [17] to view the results and quickly spot errors. A sample of the output over time for wetting and drying marshland is shown in Figure 3.

Working on a single block of code at a time simplified the process of identifying the source of the discrepancies.

Profile

After GPU results have been verified to be the same as the sequential version, the code can be further tuned to improve performance.

The process begins by identifying which sections of the program take up the largest portions of the runtime. This can be done with simple timers in the code surrounding sections of interest. It is not possible to use these timers inside of code offloaded to the GPU. Instead, Nvidia's CUDA profiler can be used to get performance metrics for the CUDA kernels including branch divergences and memory bandwidth.

For ALGE3D, we primarily used host-side timers to profile the code to determine elapsed time between launching and completing a kernel. This allowed us to identify which sections consumed the majority of the runtime and how this changed and responded to alterations of the code. In particular, adjusting which memory arrays were copied in and out before and after the kernels caused significant changes in the amount of time a particular kernel might take. We frequently revisited already tuned kernels to adjust this further.

Binaries produced by PGI's compilers are also able to notify the developer of kernel launches and respective timing information by using the ACC_NOTIFY environment

variable. This can make identifying slow memory copies much simpler.

Tweak

The vast majority of our development time was spent trying to improve runtime performance of the parallelized code. Oftentimes the automatically generated code was slower than the sequential version even for large blocks of heavy computation for which GPUs are ideally suited.

As we discovered while profiling the code, the overwhelming majority of the time spent in the GPU kernels was used to copy data to or from the host. While the compiler attempts to identify the minimum amount of copies that are needed for correct computational results, it performs the analysis on a per kernel basis, or at best per file basis if a compute region is defined across multiple loop bodies. As a result, it is unable to determine whether some data might be immediately reused on the GPU, if some computed result is temporary and not needed on the host, or if the host's value before the start of the kernel is overwritten and not used directly in the calculations on the GPU.

We achieved massive speedups by identifying these unnecessary memory transfers and eliminating them. The compiler provides directives for keeping memory resident on the GPU or for ignoring results either on the GPU or host. Using these directives, we eliminated any unneeded transfers by keeping temporary data on the GPU between kernels where they were needed and not copying it back when it was not needed in host calculations.

An example of this can be seen in Figure 4, which shows kernels in two source files that share memory references that can be combined into a single copy in and copy out. By adding the `!$acc copy` directives, `u` will only be copied in once and copied out once. Otherwise, the compiler will naively copy `u` in before each kernel then out again.

An additional source of speedup came from preallocating space on the GPU. When left to perform memory management automatically, the compiler will allocate and deallocate any memory buffers on the GPU as kernels are launched and terminated. We found that in many cases we were reusing the same buffers of memory in different kernels. It was wasteful to allocate and deallocate them between kernels. To improve on this, we instead preallocated buffers that were large enough to hold grids used by ALGE3D's calculations and instructed the compiler to copy into these buffers rather than allocate new ones.

The compiler also provides facilities for adjusting kernel scheduling such as how many threads to launch at once or how to stride through the arrays. In our experience, adjusting the loop scheduling was rarely beneficial, as the compiler selected optimal schedules automatically in most cases. Occasionally, particularly for loops with very few iterations, instructing the compiler to reduce vectorization improved performance.

Vectorization in this case refers to the process of dividing loops up into 'lanes' and running multiple iterations in a single GPU thread. This can be beneficial if there are still enough threads to make full use of the GPU's computational resources as it can reduce overhead of launching more kernels. However, for loops with very small trip counts, reducing vectorization can force more threads to be launched and improve utilization of the GPU hardware.

More specifically, we found that kernels that used trip counts on the order of tens of threads benefited from the vectorization process. These loops still performed more slowly on the GPU than on the host, but there was a net benefit for them to be on the GPU because it removed the need to copy memory back and forth from the GPU and host.

Verify

Because of changes made when tuning for performance, it is critical to verify the simulation output continuously as changes are made. Removing or consolidating memory copies can lead to significant performance gains, but if important memory is erroneously not copied when needed, the results of the simulation will be incorrect. Incorrect manual code transformations or forcing the compiler to parallelize non-parallel code can likewise lead to mathematical errors in the output.

```
alge3d.f90:
! Copy u in before either kernel is run
!$acc copyin(u)

call foo()
call bar()

! Copy u back out now that the kernels are done
!$acc copyout(u)

foo.f90:
! Subroutine signature details have been omitted
! This tells the compiler that u doesn't need copying
!$acc present(u)
!$acc region
do j=1,ny
  do i=1,nx
    u(i, j) = 2 * v(i, j)
  end do
end do
!$acc end region

bar.f90:
!$acc present(u)
!$acc region
do j=1,ny
  do i=1,nx
    u(i, j) = u(i, j) + 10
  end do
end do
!$acc end region
```

Figure 4: Manual Memory Movement

At this stage it might be useful to perform longer or more detailed simulations since the shorter verification runs done earlier may miss small accumulating errors.

Repeat

The entire process is repeated which includes selecting loop bodies to be parallelized, examining them with the compiler, verifying the simulation, profiling it, tuning to improve performance further, then verifying simulation results again.

Results

By adhering to the iterative approach outlined above, we were able to accelerate ALGE3D, an existing legacy Fortran simulation program, by more than a factor of 15x for sufficiently large and detailed simulations while maintaining accuracy compared to the sequential code. Figure 5 shows how performance compares for different data set resolutions executed on an Intel Core i7-4770K at 3.5 Ghz and a GeForce GTX Titan GPU with 2,688 cores. The values given are speedups relative to the CPU using the same data set. Measurements were taken for the global execution time for the simulation including all startup, I/O and preprocessing overhead.

Low resolution simulations perform worse on the GPU than CPU. This was caused by the overhead of launching kernels and copying data. However, the GPU-based performance rapidly increased compared to the CPU as simulation resolution increased. At a resolution of 49m² giving an increase in data size by a factor of 2.04, the GPU begins to outperform the CPU. At resolutions of 4m² or smaller which increases the data size by a factor of 25 or more, a peak speedup of 15.3x was achieved. The speedup appears to plateau at around 15x. Higher resolutions that generate even larger data sets may give further improvements in

performance, but due to a limit of 6 GB of memory on the GPU, we were unable to test under these conditions.

CONCLUSION

We have demonstrated that using directive-based code parallelization tools with PGI's CUDA Fortran accelerator significantly improve the performance of legacy code bases resulting in significant improvements in runtime performance with minimal modifications to the source code.

We have also shown that a great deal of work remains to be done with automatic memory management of accelerator devices. The vast majority of our development time was spent manually tuning these aspects of the program. Improvements in the compiler's management of these resources could provide great productivity improvements for software developers.

FUTURE WORK

While we achieved the goal of improving ALGE3D's execution time substantially with minimal changes to the source code, there exists potential for further gains by using multiple GPUs in a distributed computing environment. Doing so could also alleviate some of the boundaries we encountered such as the on-board memory limit of GPU hardware which restricts the problem size.

ACKNOWLEDGMENT

The authors acknowledge the Research Cyberinfrastructure (RCI) group (<http://www.sc.edu/rci>) at the University of South Carolina for providing high performance computing resources and technical expertise and the Savannah River National Laboratory (<http://srl.doe.gov>) for providing the hydrodynamics research application that contributed to the results of this research.

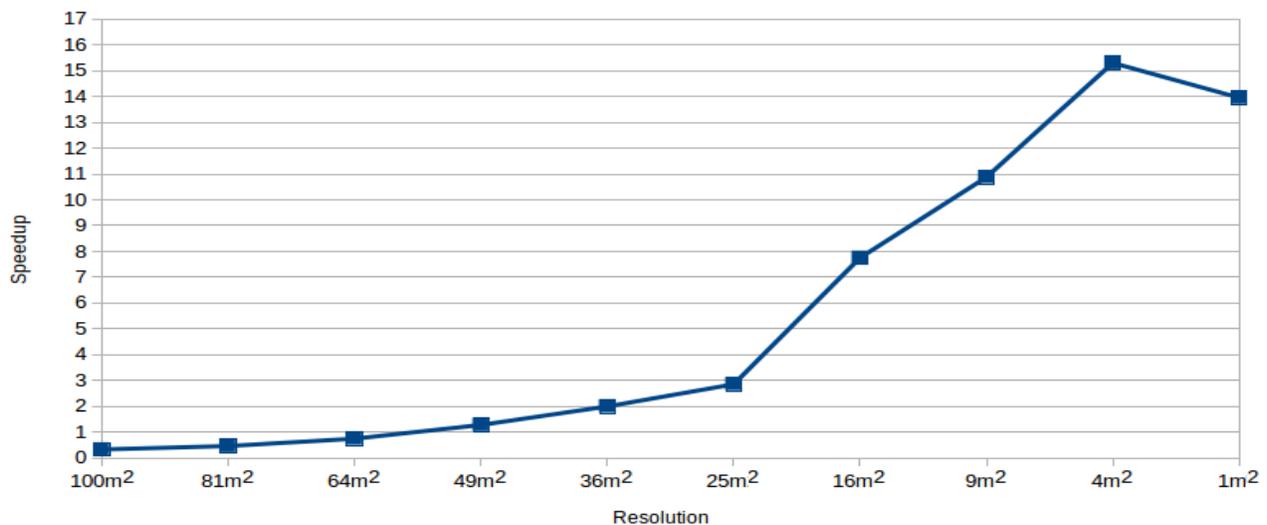


Figure 5: GPU Speedup

REFERENCES

1. "CUDA Toolkit Documentation v6.5", <http://docs.nvidia.com/cuda/index.html>, November 2014.
2. "The OpenCL Specification version 1.2", <http://www.khronos.org/registry/cl/specs/openc1-1.2.pdf>, November 2014
3. "The OpenACC Application Programming Interface version 1.0", http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf, November 2014.
4. "OpenMP Application Program Interface version 4.0", <http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf>, July 2013.
5. "PGI Accelerator Compilers With OpenACC Directives", <https://www.pgroup.com/resources/accel.htm>, November 2014.
6. Garrett, A. J. and A. L. Boni. "ALGE: A 3-D Thermal Plume Prediction Code for Lakes, Rivers and Estuaries (U)", Special Programs, Savannah River Technology Center, 1997.
7. Bella, Gino et al., "Using openMP on a hydrodynamic lattice-Boltzmann code", Proceedings of the Fourth European Workshop on OpenMP, Roma., 2002.
8. Wróblewski, Pawel, and Krzysztof Boryczko, "Parallel simulation of a fluid flow by means of the SPH method: OpenMP VS. MPI comparison", Computing and Informatics 28.1 (2012): 139-150.
9. Dong, Tingxing, et al., "Hydrodynamic Computation with Hybrid Programming on CPU-GPU Clusters", Innovative Computing Laboratory, University of Tennessee.
10. Mininni, Pablo D., et al., "A hybrid MPI-OpenMP scheme for scalable parallel pseudospectral computations for fluid turbulence", Parallel Computing 37.6 (2011): 316-326.
11. Schive, Hsi-Yu, Ui-Han Zhang and Tzihong Chiueh, "Directionally unsplit hydrodynamic schemes with hybrid MPI/OpenMP/GPU parallelization in AMR", International Journal of High Performance Computing Applications 26.4 (2012): 367-377.
12. Bernaschi, Massimo, et al., "A flexible high-performance Lattice Boltzmann GPU code for the simulations of fluid flows in complex geometries", Concurrency and Computation: Practice and Experience 22.1 (2010): 1-14.
13. Wienke, Sandra, et al., "OpenACC—first experiences with real-world applications", Euro-Par 2012 Parallel Processing, Springer Berlin Heidelberg, 2012 859-870.
14. Herdman, J. A., et al., "Accelerating hydrocodes with OpenACC, OpeCL and CUDA", High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion: IEEE 2012.
15. "IEEE 754-2008 Standard for Floating-Point Arithmetic", IEEE Std 754-2008, pp 1-70, August 2008.
16. Whitehead, Nathan, and Alex Fit-Florea, "Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs", Nvidia technical white paper (2011).
17. <http://tecplot.com>, November 2014.
18. Lee, Seyong, and Rudolf Eigenmann, "OpenMPC: Extended OpenMP programming and tuning for GPUs", Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE Computer Society, 2010.
19. Pahlevan, N., A. J. Garrett, A. D. Gerace, J. R. Schott, 2012, "Integrating Landsat-7 imagery with physics-based models for quantitative mapping of coastal waters near river discharges", Photogrammetric Engineering and Remote Sensing. 78(11), 1163-1174.
20. Blanton JO, Garrett AJ, Bollinger JS, Hayes DW, Koffman LD, Amft J, Moore T, 2010, "Transport and Retention of a Conservative Tracer in an Isolated Creek-Marsh System", Estuarine, Coastal and Shelf Science, 87(2): 333-345.
21. Blanton JO, Garrett AJ, Bollinger JS, Hayes DW, Koffman LD, Amft J, 2009, "Transport and Dispersion of a Conservative Tracer in Coastal Waters with Large Intertidal Areas", Estuaries and Coasts, 32(3):573-592.

Sharer Status-based Caching in tiled Multiprocessor Systems-on-Chip

P. Parayil Mana Damodaran, A. Zaib, S. Wallentowitz, T. Wild, A. Herkersdorf

Institute for Integrated Systems
Technische Universitaet Muenchen, Germany
preethi.parayil@tum.de, herkersdorf@tum.de

ABSTRACT

In multi-core systems with cache-to-cache forwarding, the data access latency depends heavily on the sharer status of cache-lines (CLs). The sharer status of a CL is defined by the number of copies of the CL in the whole system. Exclusive single-copy CLs have higher latency for different cache operations (e.g. misses, eviction, invalidation) when compared to shared multi-copy CLs. This is because, single-copy CL misses have to target globally shared system memory in case of a miss, while multi-copy misses may retrieve data from a CL in the neighborhood. These differences leads to dissimilarity in cache design requirements for single-copy CLs and multi-copy CLs. Unfortunately, state-of-the-art private cache designs do not consider this dissimilarity of CLs in all aspects of the cache design. This results in overhead in data access time and cache usage.

In this paper, we present a concept of dynamic partitioning of the CLs depending on their sharer status (either single-copy or multi-copy). Based on this concept, we introduce a cache hierarchy in which the hierarchy levels are made different depending on the sharer status of CLs and vary dynamically according to application requirements. In addition, we propose an optimized coherence scheme that reduces the latency overhead of single-copy CLs. The scheme also provides an automatic write-upgrade which reduces the write access time.

A cycle-accurate SystemC model of the proposed caching concept is simulated using SPLASH-2 and PARSEC benchmarks. It shows a more than 10% reduction in average memory access time compared to locality-aware adaptive cache models. It also shows a more than 40% reduction in average memory access time compared to cooperative cache models such as elastic and distributed cooperative caches where, these access latency benefits come at a cost of an on average 30% increase in network-on-chip (NoC) traffic. This once more underpins that an appropriately dimensioned on-chip interconnect is prerequisite for attaining high compute performance in large multi-core systems.

Author Keywords

Cache partitioning; cache hierarchy; distributed directory cache coherence; multiprocessor systems-on-chip;

ACM Classification Keywords

B.3.2 MEMORY STRUCTURES Design Styles: Cache memories, Shared Memory

INTRODUCTION

In shared memory-based multiprocessor systems-on-chip (MPSoC), shared data accesses are always a bottleneck forcing the entire system to become sequentialized. Large MP-SoCs are typically structured as a two dimensional array of compute tiles, memory tiles and I/O tiles. On-chip caches play a significant role in improving the performance of these systems. They reduce the average memory access time and in addition provide a parallel environment for shared data access.

On the basis of the extent of isolation of the cache storage space vis-à-vis its users (where a user could be a thread, a group of threads, a core, a group of cores or a tile), caches can be sub-divided into two categories. (a) *Shared Caches*: cache space shared by multiple users. (b) *Private Caches*: cache space accessed by a single user [11]. The cached data itself can be of two types based on the extent of sharing [11, 8, 17]. (a) *Private or non-shared data*: data which are exclusive to a single user, (b) *Shared data*: data which are read and/or written by more than one user. Both private and shared caches can store either private or shared data, or both.

In a private cache, a user stores their own copy of a CL in their respective cache space. This means that shared data in private caches have multiple copies in the system. This can induce data incoherency issues. Such issues are avoided using coherence protocols such as MSI, MESI, MOESI and MESIF [2, 14, 11], implemented using snoop or directory hardware schemes. In a shared cache, incoherency issues can never arise due to its default nature of non-isolation.

In larger MPSoC systems, both private and shared caches are placed at different hierarchy levels, e.g. the common 2-level cache hierarchy with core/tile private L1/L2 caches with address-interleaved banked last level cache (LLC) in Tile64. For better utilization of the cache storage space, especially in larger L2 and shared LLC, private and shared sections are made to share space like in Fig.1 [16, 15]. These cache systems are network-on-chip (NoC) interconnected and they implement distributed directory-based cache coherence. This paper looks into the cache usage and access latency optimization strategies in private caches of such a cache system as in Fig.1.

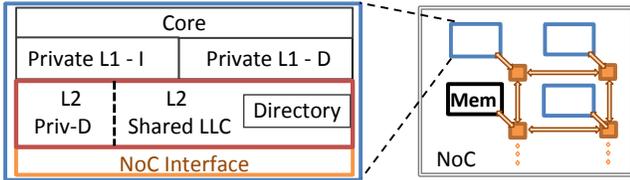


Figure 1. 2-level cache hierarchy where L2 shares space with shared LLC and private cache in a tile

Within a 2-level cache hierarchy system, shared and private data differ in multiple aspects such as access rate, reuse time and distribution of CL users [8, 18, 10]. Within the shared cache region, the shared and non-shared data are partitioned and prioritized considering their difference in access and reuse rate [8, 17, 18]. In the private cache regions, CL coherence state (i.e. I (Invalid), S (clean, shared), M (dirty, private), E (clean, private), F (clean, shared) and O (dirty, shared)) classifies the data based on the properties clean/dirty and shared/private [2, 14]. However, these classifications are not correspondingly mapped to all the design parameters (especially the cache hierarchy and the cache coherence management). Uniform cache design for dissimilar types of data causes high latency overhead.

Multiple recent research studies [10, 13, 15] have come up with complex solutions to handle this crucial issue. Here we propose a solution which has very low area overhead. Our solution specifically focuses on the private caches with cache-to-cache (C-C) data forwarding support [2, 13, 14]. In MP-SoC models such as Tiler64, it is observed that the main-memory/shared LLC accesses are 30-50 times slower than L1 accesses. By facilitating coherent cache-to-cache shared data transfers among smaller private caches, C-C support can reduce these high latency accesses.

Considering the cache operations of shared and exclusive CLs in C-C forwarding-supported private caches as in Fig.2, 3 points can be noted: (1) Latency of the cache operations is different for shared and non-shared CLs. (2) Read/write miss, invalidation and eviction can cause additional overhead on non-shared CLs. This is mainly due to high latency transfers to the shared LLC or memory resources with higher probability for contention. (3) On the other hand, write upgrades can cause overhead on shared CLs due to the multiple number of sharers to be invalidated. These differences point to the latency overhead in state-of-the-art designs, caused just because the cache management is unaware of the degree of data sharing.

These observations triggered our curiosity to investigate whether a cache management method that treats single-copy CL differed from multi-copy CL is advantageous. In this paper, we propose a novel idea of partitioning private caches on the basis of the sharer status of the CL. The proposed concept classifies the CLs as **single-copy cache-line (SCCL)**: representing exclusive CLs which contain non-shared exclusive data and **multi-copy cache-line (MCCL)** representing CLs with more than one copy containing shared data. We also come up with an optimized cache hierarchy and coherence policy for each of these types of CLs, which are adaptive over time according to application requirements. With these

	Multi-copy shared CLs	Single-copy exclusive CLs
Read miss	C-C fwd delay	Main-memory/shared LLC access delay
Write miss	C-C fwd delay	Main-memory/shared LLC access delay
Write upgrade	Sharer invalidation delay	Direct write access; no coherence delay
Invalidation	Simple invalidation no coherence delay	C-C fwd delay or delay for write-back to shared LLC/main-memory
Eviction	Simple invalidation no coherence delay	2-hop eviction confirmation delay and/or delay for write-back to LLC/main-memory
	Low Latency	High Latency

Figure 2. Comparison of cache operations of shared multi-copy and exclusive single-copy CLs in private L1/L2 caches

changes, we bring in two major enhancements to optimize the cache usage and to minimize the data access time: (1) Provide suitable hierarchy levels for CLs according to their respective sharer status. This facilitates optimized placement of high locality CLs nearest to the requester for a longer duration. (2) Minimize the coherence overhead of SCCLs and provide a simple method to dynamically switch between single-copy and multi-copy partitions.

The rest of the paper is organized as follows. Motivation and the related research is presented in section II. The proposed caching concept is described in section III. Section IV presents the details of the simulation and results. Finally, section V concludes the paper.

MOTIVATION AND RELATED WORK

In private caches with C-C dirty or clean data forward support [14, 13, 12, 9], multi-copy and single-copy CLs show a vast difference in latency in various cache operations as shown qualitatively in Fig.2. It can be observed that SCCLs suffer latency overhead in all cases except the write-upgrade operation. This is mainly due to 3 reasons: (1) SCCLs have only one copy within the system, whereas MCCLs have many. (2) If evicted, SCCLs must be re-fetched from the slow shared LLC or main-memory. (3) MCCLs support C-C forwarding of data; this has the same effect as that of an additional (virtual) cache hierarchy between L1 and the shared LLC or memory.

Now we consider the data requirements of shared memory parallel applications, e.g. NAS applications such as bt, lu, ft, cg; RMS benchmark applications such as face, spg; and commercial workloads such as jbb, apache [1, 5, 3]. It has been observed by multiple state-of-the-art studies [5, 8, 17, 4] that around 60% of loads/stores of the applications access exclusive copy CLs. This implies that the CL is holding either private data or dirty/clean exclusive shared data. In Fig.3, we show the distribution of CL accesses of SPLASH-2 and PARSEC benchmark applications. It can be observed that for most of the shared memory parallel applications,

$$(\text{count of SCCL accesses}) > (\text{count of MCCL accesses}).$$

Now, recalling the equation of average memory access time (AMAT) Eq.1, AMAT of the total application is the proportional sum of AMAT of the single-copy and multi-copy CL accesses. Hence,

$$AMAT = Rate_{hit} * Time_{hit} + Rate_{miss} * Time_{miss} \quad (1)$$

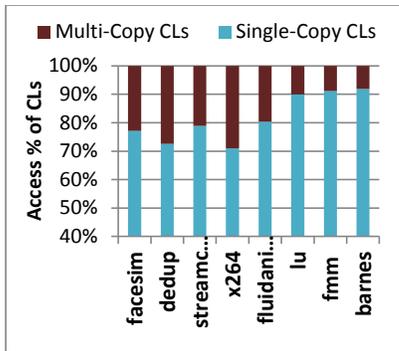


Figure 3. Distribution of single-copy/multi-copy CL accesses in PARSEC & SPLASH-2 benchmarks

$$AMAT_{app} = P_{SC} * AMAT_{SC} + P_{MC} * AMAT_{MC} \quad (2)$$

where, P_{SC} and P_{MC} represent the access ratios of single-copy and multi-copy CLs.

From Fig.2, Fig.3, Eq.2 and from the above observations, two main remarks can be made. (1) In C-C forward enabled systems, $AMAT_{SC} > AMAT_{MC}$. (2) For most of the shared-memory parallel applications, $P_{SC} > P_{MC}$; this implies $AMAT_{app} \approx AMAT_{SC}$. Hence, intuitively, $AMAT_{app}$ can be minimized by reducing $AMAT_{SC}$ with minimum penalty on the multi-copy CLs.

Our caching concept optimizes the $AMAT_{app}$ by partitioning the CLs on the basis of the sharer status (single-copy/multi-copy), depending on the application requirements at run-time. We propose:

(1) a dynamic and adaptive cache hierarchy which provides an additional hierarchy level for high latency SCCLs when compared to MCCLs. This increases the hit rate of SCCLs and decreases the L1 miss penalty. Hence from Eq.1 and Eq.2, $AMAT_{SC}$ and $AMAT_{app}$ are minimized.

(2) an automatic write-upgrade facility to the regular C-C forward enabled coherency protocols. This feature reduces the write-upgrade time of the MCCLs by converting them to SC-CLs with read/write permission. This in effect reduces the total coherency overhead and so the $AMAT_{app}$.

To regulate the penalty on MCCLs, we use the least recently used (LRU) eviction policy in L1 and L2 caches. CLs (of the type single-copy or multi-copy) which have higher temporal locality and lower reuse time are retained within the cache with higher probability. This ensures adaptive cache space usage for single- and multi-copy CLs.

Related Work

Current research deals with CL partitioning and access time optimization in mainly three different ways: (1) Operating system (OS)-based shared and non-shared data partitioning [10] (2) Cooperative caching of single-copy CLs [12, 13, 9] (3) Victim caching for core-private caches [15, 20].

R-NUCA [10] is an OS-based data partitioning approach, which requires an environment that supports memory virtualization. For R-NUCA, the pages are tagged as shared or

private according to the ratio of shared and nonshared data in a page (where ratio information is provided by the application). As R-NUCA is a completely software solution, this cannot be compared with our proposed concept, which is a purely hardware solution.

In cooperative cache architectures such as DCC [13], ECC [12], Data-Sliding [9] etc., the single-copy CLs are categorized as singlets. These singlets, when spilled, are cooperatively stored within neighboring caches. This creates an additional (virtual) cache level for singlets in comparison to non-singlet CLs. However, in comparison to the proposed concept, (1) cooperative caches have huge area overhead for the selection of the neighboring cache and respective coherence control. (2) There is no system-shared LLC, hence the miss access latency is high. (3) They provide a fixed priority LRU-based eviction scheme in the L2 cache, allotting always higher priority to singlets in comparison to locality- and reuse-based eviction as in our design.

Approaches such as victim caching and locality-aware adaptive caching [20, 16, 15] implement victim caches for all private L1 spills, where the victim cache space is shared with the shared-LLC, similar to our proposed approach. However, in comparison to our proposed concept, (1) victim caches can contain multi-copy CLs, which (a) pollutes the victim storage space, reducing the space for single-copy CLs (b) increases the count of copies of multi-copy CLs, causing larger overhead for multi-copy write-upgrade; (2) Locality-aware coherence policy requires much higher area overhead for the adaptive data storage facility; (3) Victim-based designs lack the automatic write-upgrade facility. The following section describes the proposed caching concept and design in detail.

SHARER STATUS-BASED CACHE CONCEPT

Sharer status-based Caching (Sh-C) concept brings forward the following new ideas for cache-line partitioning:

(1) A cache which classifies the CLs of a private cache into either single-copy or multi-copy types, according to its sharer status. (2) Provide a cache hierarchy specific to the sharer status of the CL (i.e. one additional hierarchy level to SCCLs in comparison to MCCLs). (3) Allow CLs to switch between single-copy and multi-copy statuses dynamically at run-time. (4) Provide automatic write-upgrade facility via directory updates.

Absorbing these optimization ideas, the Sh-C concept introduces a hierarchy scheme and a coherence policy, which are described in the sections below. Here, we consider only the private data caches, i.e. the hierarchy of caches which require coherence management. A CL is the minimum size of data that is monitored and updated; hence it can be considered as the minimum granularity to monitor the sharer status.

Cache Hierarchy:

The proposed cache has two levels in the hierarchy, similar to cooperative and victim cache reference models [20, 15, 13, 12, 9] as shown in Fig.1. However, in Sh-C private caches, the CLs are logically partitioned into single-copy and multi-copy

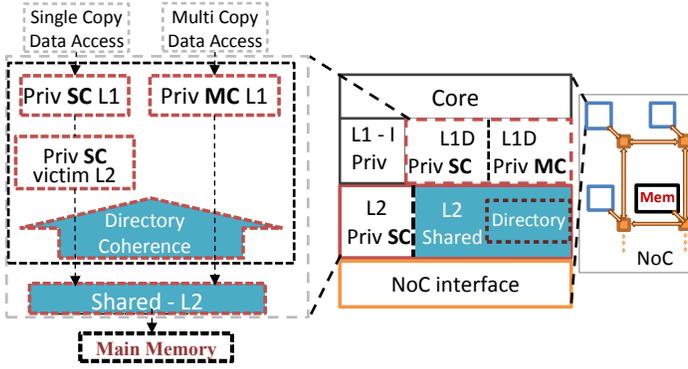


Figure 4. Logical view of cache hierarchy in the proposed Sh-C approach

types. A logical representation of the proposed hierarchy is shown in Fig.4.

Level 1: The L1 cache is logically divided into two partitions, one section with SCCLs (*Priv SC*) and the second with MCCLs (*Priv MC*).

Level 2: The L2 is space shared by two different cache sections: (a) *Private SC victim section*: victim cache for private single-copy L1 CLs. (b) *Shared L2 section*: is a statically distributed system-shared last level cache (LLC), placed logically below L1 and victim L2 in the hierarchy.

From the above description of the architecture and from Fig.4, the following points can be noted.

1. SCCLs follow a three level cache hierarchy: private L1 SC section → private victim L2 section → shared L2 section → main-memory.
2. MCCLs follow a two level cache hierarchy: private L1 MC section → shared L2 section → main-memory.
3. The coherence of L1 cache and Private SC victim section of L2 cache is managed by distributed directories.

In Sh-C, SCCLs have more levels in the hierarchy. This implies that they have higher probability for load/store hit within the requesting tile. Hence from Eq.1, $Rate_{miss}$ is decreased. This reduces the $AMAT_{SC}$. In addition, it should be noted that the status of a CL can change dynamically from SCCL to MCCL or vice-versa according to the access patterns of the application. In such scenarios, a particular CL still follows the most suitable hierarchy at run-time. That is, a CL follows a two-level hierarchy when there are multiple copies of itself; the same CL follows a three level hierarchy when there is only a single-copy in the system, thus optimizing the access in both cases. How the coherence management facilitates a convenient switch between sharer statuses is explained in the section below.

Cache Coherence:

The proposed design of the Sh-C concept uses a modified version of the MESIF protocol [14, 13, 12, 2] with an automatic write-upgrade facility. The L1 and L2 private victim cache sections are coherence managed using this protocol. It has states designed on the basis of the sharer status as shown below:

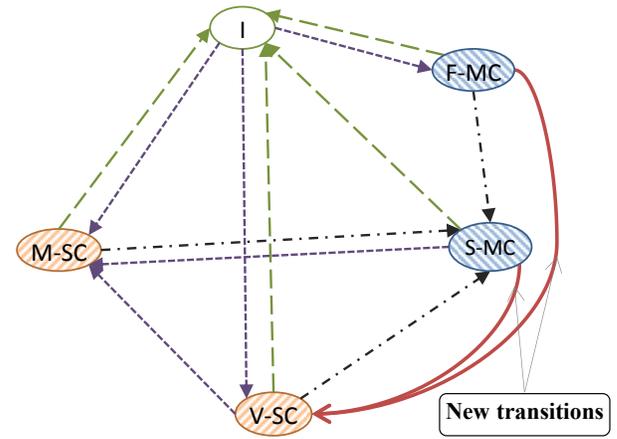


Figure 5. Representation of the proposed cache-coherence

1. I (Invalid)
2. M-SC (modified single-copy)
3. V-SC (valid single-copy)
4. S-MC (shared multi-copy)
5. F-MC (forward multi-copy)

The major difference between the proposed coherence method and MESIF are: (a) Sh-C coherence method utilizes the sharer status along with coherence states. (b) The sharer status enables the coherence protocol to add two new state transitions, namely Shared-MC to Valid-SC as well as Forward-MC to Valid-SC. The state diagram of the proposed new protocol is shown in Fig.5 with states and state-transitions representing their respective coherence information. These state transitions facilitate the CLs to reach back to states with **single-copy (SC)** status from the corresponding states with **multi-copy (MC)** status, when all the other copies of the same CL are evicted or invalidated. This helps to preserve the CL within the system for longer at a location nearest to the requester of the CL. Thus, the Sh-C concept facilitates the novel idea of automatic sharer status switching in CLs (from multi-copy status to single-copy status) which is not possible in any of the state-of-the-art cache coherence protocols.

This write-upgrade notification is initiated by the directory controller. For every eviction notification received at the directory, an automatic sharer status assertion check is done. If the status changes from MC to SC (which is the change in sharer status during write-upgrade), a sharer status change notification is sent from the directory to the only CL sharer. Following the notification, the CL sharer upgrades its status from MC to SC. Hence MC to SC status transitions provide an automatic write-upgrade facility outside the load/store critical path and hence can optimize the AMAT of the system.

This concept can also be embedded into other cache protocols. For instance, the MOESI protocol [2] can incorporate the single-copy or multi-copy CL partitioning. This will add three additional state transitions, S to E, O to E and O to M, which allow the transition from multi-copy CL status to single-copy.

Implementation:

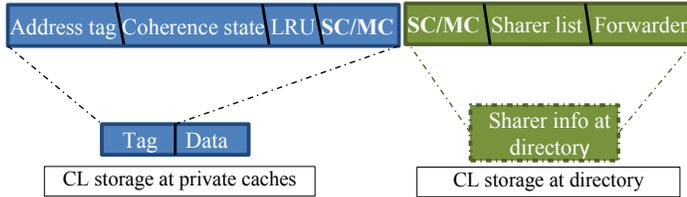


Figure 6. Modified cache-line representation

	L1 multi-copy CLs	L1 single-copy CLs
Read miss	C-C fwd delay	Delay for L2 private victim read
Write miss	C-C fwd delay	Delay for L2 private victim read
Write-upgrade	If multi-shared, sharer-invalidation else, automatic write-upgrade	Direct write access; no coherence delay
Invalidation	Simple invalidation; no coherence delay	Delay for C-C data forward and self-invalidation
Eviction	Simple invalidation; no coherence delay	Delay for write-back to L2 private victim
	Low Latency	High Latency

Figure 7. Comparison of cache operations of single- and multi-copy CLs in Sh-C approach

In Sh-C implementation, CLs of private caches carry a 1 bit sharer status information in the tag. The value of the 1 bit sharer status, SC or MC, is stored as shown in Fig.6. To observe the proposed latency optimizations, cache operations on single-copy and multi-copy CLs are presented in Fig.7, which can be directly compared to the state-of-the-art protocols in Fig.2.

The sharer status is a very important parameter in the cache-lookup, coherence management and eviction data redirection decisions. Hence, this information is stored and controlled by the directory as shown in Fig.6. Since the sharer status directly implies the read and write permission for the CLs, CL coherence state information is not stored in the directory. Directory stores only the sharer status and the list of sharers.

Our design of the Sh-C concept has all its private caches i.e. L1 and private victim section of L2 coherence managed by the proposed modified MESIF protocol. Finally, the shared L2 section is a simple MSI write-back cache; where the read misses and write-backs directly hit the main-memory. Sharer status is passed to the caches along with the coherence messages such as invalidation, eviction/read/write acknowledgments and write-upgrade notification. Excluding the interaction between core and L1, all other internal transactions (data and coherence messages) in the private cache hierarchy include the sharer status MC/SC as an extra bit of control data.

Overhead Analysis

The design of the Sh-C concept reduces AMAT of the system at the cost of very small area and traffic overhead.

(1) Area: The Sh-C concept adds 1 bit SC/MC sharer status information in each CL tag of private caches and directory. However, coherence states are no longer stored in the directory as shown in Fig.6. Hence, the overall increase in area is negligible.

(2) NoC traffic: The Sh-C adds a new coherence command write-upgrade notification from directory to caches during MC to SC transitions, thus introducing additional traffic. However, the increase in hit rate of SCCLs due to single-copy victim caching reduces the single-copy write-backs to shared LLC and the shared LLC write-backs to memory. This reduces the total write-back traffic. Hence, the overall traffic overhead depends on the ratio of the above two effects in reality.

In the current design of the Sh-C concept, the distinction between SC and MC requests cannot be done from the core or application side. Hence, all L1 misses must go through a three-level hierarchy, causing three cycles additional delay for all MC read-misses. However, in the future, we plan to provide an application-level or a speculative hardware middle-layer support between core and L1.

Detailed simulation-based analysis of the Sh-C concept is presented in the following section.

SIMULATION

Architectural Parameter Value	
Number of cores	15 @ 1 GHz
Compute pipeline per core	Out-of-order, single-issue, < 3 active requests
Consistency	Sequential consistency support for s/w fences
Physical address length	32 bits
ISA of cores	ALPHA
Cache sub-system	
L1 cache per core	16KB, 2-way set associative, 2 cycles for hit
L2 cache per tile	128KB, 4-way set associative, 5 cycles for hit
Cache line size	32 bytes
Coherence scheme	Distributed directory coherence
Eviction policy	LRU policy
DRAM read-write delay	300 cycles
NoC sub-system	
Topology	Mesh
Routing	X-Y routing
NoC size	4x4
Hop latency	3 NoC cycles = 3 cycles (1-router, 1-link, per packet)
Size of packet without data	2 flits (header+address)
Size of packet with data	10 flits (header+address + data)
Benchmark applications	
SPLASH-2 benchmark apps	LU, FMM, BARNES, OCEAN
PARSEC benchmark applications	Blackscholes, canneal, dedup, facesim, rtview, streamcluster, fludanimate, bodytrack, x264, vips

Figure 8. Architectural details of simulation system

SPLASH 2	
LU	512 x 512 matrix, 16 x 16 blocks
BARNES	16K particles
OCEAN	258 x 258 ocean
FMM	512 molecules
PARSEC	
BLACKSCHOLES	4k options
BODYTRACK	1 frame 1000 particles
CANNEAL	10000 elements
DEDUP	31MB medias.dat
FACESIM	standard small simulation
FLUIDANIMATE	5 frames, 35K particles
RTVIEW	3 frames of "480x270"
STREAMCLUSTER	4096 points per blocks, 1 block
VIPS	im_benchmark, 1 frame, 1600x1200
x264	eledream_640x360_8 with qp 20, partitions b8x8,i4x4

Figure 9. Benchmark application input set

Simulation System

Simulations were carried out on a cycle-accurate SystemC model of a tiled MPSoC system. Each tile in the NoC of the simulator includes a processor, L1 and L2 cache and a directory. The tiles which carry memory do not include a core or L1. Similarly, the tiles which carry computation elements do not carry a memory unit. This simulator models the actual transaction of data packets. It requires the memory system (including the cache hierarchy) to be functionally correct in order to complete the execution of the simulation. The table in Fig.8 describes the details of the simulation setup.

The Gem5 simulator [7] was used for generating application-specific trace files which were fed as input to a SystemC multi-core model. We have used four SPLASH-2 [19] benchmarks and ten PARSEC [6] benchmarks, Fig.9. We have successfully run SPLASH-2 and PARSEC benchmarks to completion. This is a good test which assures the correctness of the implemented protocols.

For the Sh-C concept evaluation, we have used three reference models distributed cooperative caches (DCC) [13], elastic cooperative caches (ECC) [12] and locality-aware adaptive cache (LOC-ACC) [15]. The models are implemented without any additional software or virtualization support for the cache partitioning or placement.

Results

The Sh-C model is evaluated with reference to cooperative and victim cache models, using three metrics (1) average memory access time, (2) cache hit/miss distribution and (3) NoC traffic. For easy analysis of the results, the proposed model is marked in red color in all the graphs.

(a) Analysis of Memory Access Time:

We consider the AMAT of the read/write requests for each of the benchmarks after the cache-initialization period ($\sim 10\%$ of the total simulation time). The AMAT values of all models are normalized with Sh-C AMAT values as presented in Fig.10. Fig.10 clearly shows that in comparison to Sh-C, the LOC-ACC model and the cooperative cache models ECC

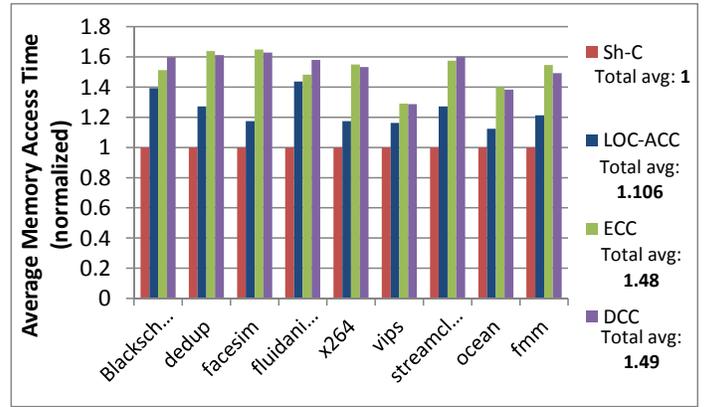


Figure 10. Distribution of Average Memory Access Time per Request

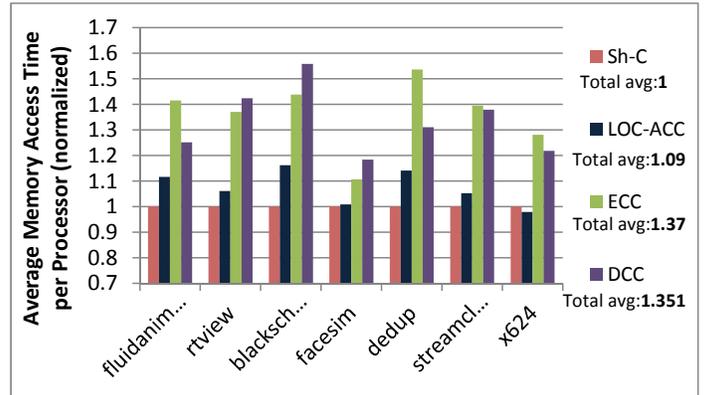


Figure 11. Distribution of Average Memory Access Time per Processor (including cache initialization time, $\sim 10\%$ of the total simulation time)

and DCC take 10% and 45% more time respectively for data accesses. However, it should also be noted that the system-shared LLC can increase the time taken for first read and write accesses by up to 18-20%. To observe the effect of these longer first-time read/writes, the average memory access time per processor for the entire simulation time is collected and averaged in Fig.11. It can be observed from Fig.11 and Fig.10 that when first-time data accesses are considered, the average memory access time of the LOC-ACC and the Sh-C model increases by around 6-10%.

(b) Analysis of cache performance (hit/miss):

Cache sub-system performance is evaluated using hit rates of L1, L2 and main-memory in the following manner and is shown in Fig.12.

(a) L1 hit rate

(b) Node hit rate: L1 and L2 victim section hits do not require a NoC interaction. Hence, they are named as node hit.

(c) On-chip hit rate: L1, L2 victim section and L2 shared section accesses do not involve off-chip memory. Hence, they are named as on-chip hit.

(d) Off-chip rate: represents all the off-chip main-memory accesses. This includes the read-write misses as well as the write-backs from the shared LLC.

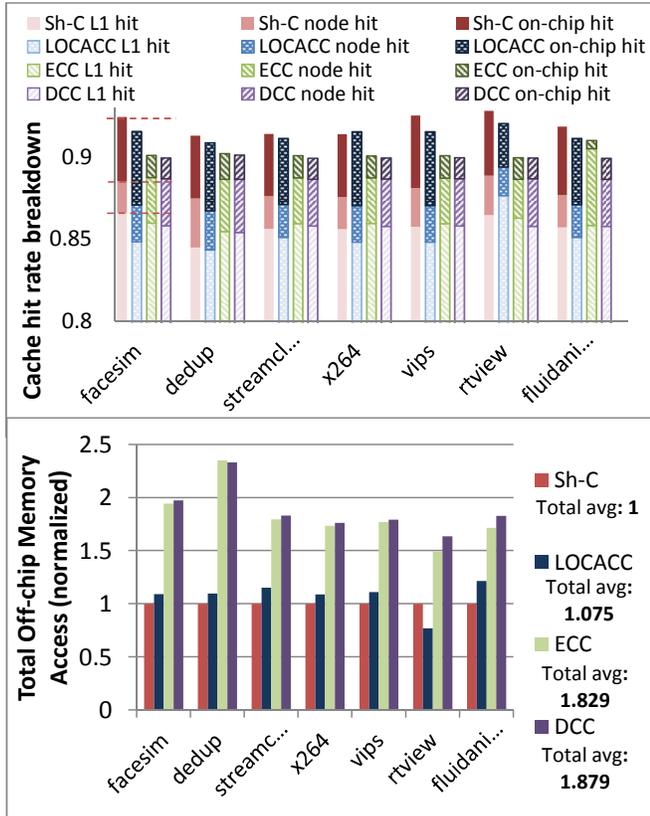


Figure 12. Cache hit rate distribution and memory-access comparison

From Fig.12, it can be observed that L1 and node hit rates of the Sh-C model are less than the DCC-ECC models. This is expected, since in Sh-C, the L1 and L2 caches hold partitions also for MCCLs and system-shared LLC respectively. Hence, the effective size of private CLs in L1 and L2 caches

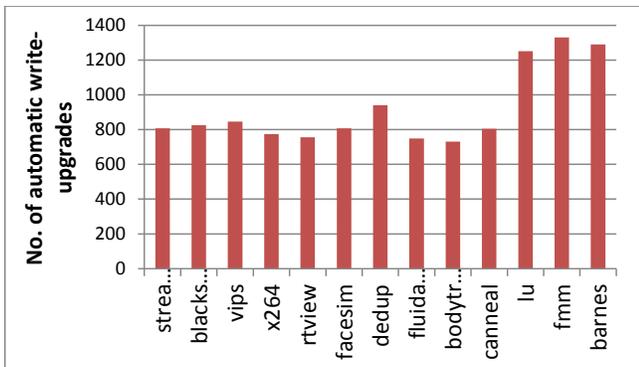


Figure 13. Automatic write-upgrade notifications in Sh-C model

is reduced. Nevertheless, the system-shared L2 section of the Sh-C model is able to reduce the off-chip main-memory accesses, satisfying most of the requests at the shared LLC. It can be noted that L1 and node hit rates of the Sh-C model are higher than LOC-ACC model. This directly points to the advantage of partitioning L1 and private victim L2 as SC-CLs and MCCLs. Similar behavior is reflected in the main-memory accesses of models in Fig.12.

The increased on-chip hit rate of the Sh-C model could be due to two main optimizations in Sh-C. (1) The SC/MC partitioning and increase in SCCL hierarchy levels; this is directly observable as node hit in Fig.12. (2) The new automatic write-upgrade which is facilitated by our model. Fig.13 represents the total count of automatic write-upgrades in the Sh-C model in multiple benchmark applications.

(c) Analysis of NoC traffic:

To evaluate the effect of the NoC traffic, the count of flits traversed through the NoC during the simulation is compared. The NoC traffic rate of the Sh-C and the reference models is presented in Fig.14. A traffic increase of on average 30% is

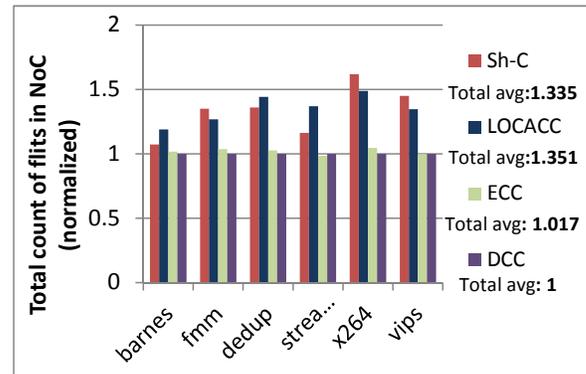


Figure 14. Network-on-Chip traffic comparison

observed in the Sh-C in comparison to DCC and ECC models. This can be expected due to two main reasons. (1) During first time read-writes as well as during shared L2 misses, the system-shared LLC in the Sh-C and LOC-ACC models in comparison to ECC-DCC models causes multiple additional flit travels. (2) The additional coherence state changes in the proposed Sh-C model require directory interactions, causing an increase in the NoC traffic. Therefore, MPSoC architecture design has to be aware of this price for higher compute performance in large multi-core systems and consider it during NoC link rate and NoC router dimensioning.

CONCLUSION

In this paper, we presented a novel concept of dynamic fine-grained cache partitioning of private caches on the basis of the sharer status of cache-lines, i.e. single-copy CL or multi-copy CL. Based on this, we introduced a complete cache design consisting of a dynamic cache hierarchy and a cache coherence policy. In the proposed design, the CLs in private caches are logically partitioned as single-copy and multi-copy CLs. A CL is allowed to dynamically switch between partitions according to the application access patterns. In order to optimize the average memory access time, the single-copy CL partition (which requires more main-memory accesses) is designed with an additional hierarchy level. We also introduced a cache coherence policy which is facilitated with automatic write-upgrade. Thus, the average memory access time of each of these partitions is optimized, and adapted dynamically to the application requirements. The concept has been evaluated using cycle-accurate SystemC models executing SPLASH-2

and PARSEC benchmark applications. From the simulation results it was observed that the Sh-C model reduces the average memory access time by 10% and 40% in comparison to locality-aware adaptive cache and cooperative cache models, respectively. These latency reductions come at the cost of a 30% increase in NoC traffic due to higher rates of cache-to-cache forwarding.

In future, we will investigate the benefits of our Sh-C method on MPSoC architectures with multiple cores per compute tile and on-chip memory tiles mapped into the shared address space.

ACKNOWLEDGMENTS

This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center “Invasive Computing” (SFB/TR 89).

REFERENCES

1. Abandah, G. *Characterizing Shared-memory Applications: A Case Study of NAS Parallel Benchmarks*. Hewlett Packard Laboratories, 1997.
2. Albonesi, D. H., Martonosi, M., August, D. I., and Martínez, J. F., Eds. *42st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42 2009), December 12-16, 2009, New York, New York, USA*, ACM (2009).
3. Aslot, V., Domeika, M. J., Eigenmann, R., Gaertner, G., Jones, W. B., and Parady, B. Specomp: A new benchmark suite for measuring parallel computer performance. *WOMPAT '01* (2001), 1–10.
4. Barrow-Williams, N., Fensch, C., and Moore, S. A communication characterisation of splash-2 and parsec. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC), IISWC '09* (2009), 86–97.
5. Beckmann, B. M., Marty, M. R., and Wood, D. A. Asr: Adaptive selective replication for cmp caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 39* (2006), 443–454.
6. Bienia, C., Kumar, S., Singh, J. P., and Li, K. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, PACT '08* (2008), 72–81.
7. 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., and Wood, D. A. The gem5 simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7.
8. Chen, Y., Li, W., Kim, C., and Tang, Z. Efficient shared cache management through sharing-aware replacement and streaming-aware insertion policy. In *IPDPS, IEEE* (2009), 1–11.
9. Dahmani, S., Cudennec, L., and Gogniat, G. Introducing a data sliding mechanism for cooperative caching in manycore architectures. In *Proceedings of IPDPSW* (2013), 335–344.
10. Hardavellas, N., Ferdman, M., Falsafi, B., and Ailamaki, A. Reactive nuca: Near-optimal block placement and replication in distributed caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture, ISCA '09* (2009), 184–195.
11. Hennessy, J. L., and Patterson, D. A. *Computer Architecture - A Quantitative Approach* (5. ed.). Morgan Kaufmann, 2012.
12. Herrero, E., González, J., and Canal, R. Elastic cooperative caching: an autonomous dynamically adaptive memory hierarchy for chip multiprocessors. In *ISCA* (2010), 419–428.
13. Herrero, E., González, J., and Canal, R. Distributed cooperative caching: An energy efficient memory scheme for chip multiprocessors. *IEEE Trans. Parallel Distrib. Syst.* 23 (2012), 853–861.
14. Hum, H. H. J., and Goodman, J. R. Forward state for use in cache coherency in a multiprocessor system. Patent US 6,922,756 B2, Intel Corporation, July 2005.
15. Kurian, G., Khan, O., and Devadas, S. The locality-aware adaptive cache coherence protocol. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 523–534.
16. Lodde, M., Flich, J., and Acacio, M. E. Dynamic last-level cache allocation to reduce area and power overhead in directory coherence protocols. In *Proceedings of the 18th International Conference on Parallel Processing, Euro-Par'12* (2012), 206–218.
17. Natarajan, R., and Chaudhuri, M. Characterizing multi-threaded applications for designing sharing-aware last-level cache replacement policies. *2013 IEEE International Symposium on Workload Characterization (IISWC) 0* (2013), 1–10.
18. Panda, B., and Balachandran, S. Csharp: Coherence and sharing aware cache replacement policies for parallel applications. In *SBAC-PAD, IEEE* (2012), 252–259.
19. Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., and Gupta, A. The splash-2 programs: Characterization and methodological considerations. In *Proceedings of the 22Nd Annual International Symposium on Computer Architecture, ISCA '95* (1995), 24–36.
20. Zhang, M., and Asanovic, K. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, 336–345.

Accelerating the LOBPCG method on GPUs using a blocked Sparse Matrix Vector Product

Hartwig Anzt
University of Tennessee
hanzt@icl.utk.edu

Stanimire Tomov
University of Tennessee
tomov@icl.utk.edu

Jack Dongarra
University of Tennessee
dongarra@icl.utk.edu

ABSTRACT

This paper presents a heterogeneous CPU-GPU implementation for a sparse iterative eigensolver – the Locally Optimal Block Preconditioned Conjugate Gradient (LOBPCG). For the key routine generating the Krylov search spaces via the product of a sparse matrix and a block of vectors, we propose a GPU kernel based on a modified sliced ELLPACK format. Blocking a set of vectors and processing them simultaneously accelerates the computation of a set of consecutive SpMV's significantly. Comparing the performance against similar routines from Intel's MKL and NVIDIA's cuSPARSE library we identify appealing performance improvements. We integrate it into the highly optimized LOBPCG implementation. Compared to the BLOBEX CPU implementation running on two eight-core Intel Xeon E5-2690s, we accelerate the computation of a small set of eigenvectors using NVIDIA's K40 GPU by typically more than an order of magnitude.

ACM Classification Keywords

G.1.3 Numerical Analysis: Numerical Linear Algebra—*Eigenvalues and eigenvectors*

Author Keywords

LOBPCG eigensolver, GPU acceleration, SpMV, SpMM

INTRODUCTION

The main challenges often associated with numerical linear algebra are the fast and efficient solution of large, sparse linear systems, and the appertaining eigenvalue problems. While linear solvers often serve as a backbone of simulation algorithms based on the discretization of partial differential equations, eigensolvers play a central role, e.g., in quantum mechanics, where eigenstates and molecular orbitals are defined by eigenvectors, or principal component analysis. With increasing system size and sparsity, dense linear algebra routines, usually based on direct solvers like LU factorization, or, in the case of an eigenvalue problem, Hessenberg decomposition [38], become less suitable as the memory demand and computational cost may exceed the available resources. Iterative methods providing solution approximations often become the method of choice. However, as their performance is, at least in case of sparse linear systems, usually memory bound, leveraging the computing power

of today's supercomputers, often accelerated by coprocessors like graphics processing units (GPUs), becomes challenging.

While there exist numerous efforts to adapt iterative linear solvers to coprocessor technology, sparse eigensolvers have so far remained outside the main focus. A possible explanation is that many of those combine sparse and dense linear algebra routines, which makes porting them to accelerators more difficult. Aside from the power method, algorithms based on the Krylov subspace idea are among the most commonly used general eigensolvers [38]. When targeting symmetric positive definite eigenvalue problems, the recently developed Locally Optimal Block Preconditioned Conjugate Gradient method (LOBPCG, see [27]) belongs to the most efficient algorithms. LOBPCG is based on maximizing the Rayleigh Quotient, while taking the gradient as the search direction in every iteration step. Iterating several approximate eigenvectors, simultaneously, in a block in a similar locally optimal fashion, results in the full block version of the LOBPCG. Applying this algorithm efficiently to multi-billion size problems served as the backbone of two Gordon-Bell Prize finalists that ran many-body simulations on the Japanese Earth Simulator [42][41]. One of the performance-crucial key elements is a kernel generating the Krylov search directions via computing the product of the sparse system matrix and a set of vectors. With the sparse matrix vector product performance traditionally limited by memory bandwidth, LOBPCG, depending on this routine, has for a long time been considered unsuitable for GPU acceleration.

In this paper we present an LOBPCG implementation for graphics processing units able to efficiently leverage the accelerator's computing power. For this purpose, we employ a sophisticated sparse matrix data layout, and develop a kernel specifically designed to efficiently compute the product of a sparse and a tall-and-skinny dense matrix composed of the block of eigenvector approximations. As this kernel also is an integral part of other block-Krylov solvers, the significance of its performance carries beyond the example integration into LOBPCG we present in this paper. We benchmark the routine against a similar implementation provided in Intel's MKL [1] and NVIDIA's cuSPARSE [35] library, and analyze the improvement it renders to the performance of the LOBPCG GPU implementation. Finally, we also compare it to the state-of-the-art multi-threaded CPU implementations of

LOBPCG based on the BLOPEX [24] code for which the software libraries *PETSc* and *hypre* provide an interface [28]. For matrices taken from the University of Florida Matrix Collection, we achieve significant acceleration when computing a set of the respective eigenstates.

RELATED WORK

Blocked sparse matrix vector product: As there exists significant need for blocked sparse matrix vector products, NVIDIA’s cuSPARSE library provides this routine for the CSR format [35]. Aside from a straightforward implementation assuming the set of vectors being stored in column-major order, the library also contains an optimized version taking the block of vectors as row-major matrix as input, that can be used in combination with a preprocessing step transposing the matrix to achieve significantly higher performance [34]. The blocked sparse matrix vector product we propose in this paper not only outperforms the cuSPARSE implementations for our test cases, but the detailed description also allows porting it to other architectures.

Orthogonalizations for GPUs: Orthogonalization of vectors is a fundamental operation for both linear systems and eigenproblem solvers, and many applications. There has been extensive research on both its acceleration and stability. Besides the classical and modified Gram-Schmidt orthogonalizations [19] and orthogonalizations based on LAPACK (xGEQRF + xUNGQR) [4] and correspondingly MAGMA for GPUs [22][39], recent work includes communication-avoiding QR [16], also developed for GPUs [5][3]. For tall and skinny matrices these orthogonalizations are in general memory bound. Higher performance, using Level 3 BLAS operations, is also possible in orthogonalizations like the Cholesky QR or SVD QR, but they are less stable (error bounded by the square of the condition number of the input matrix). These were developed for GPUs in MAGMA, including a mixed-precision Cholesky QR that removes the square by selectively using higher than the working precision arithmetic [43] (also applied to a CA-GMRES for GPUs).

For the LOPBPCG method, the most time consuming operation after the SpMM kernel is the orthogonalization (two orthogonalizations of m vectors per iteration, see Section).

LOBPCG implementations: The BLOPEX package maintained by A. Knyazev may be considered as state-of-the-art for CPU implementations of LOBPCG, as the popular software libraries PETSc and hypre provide an interface [28]. Also Scipy [23], octopus [14] and Anasazi [8] part of the Trilinos library [20] feature LOBPCG implementations.

The first GPU implementation of LOBPCG is from 2011 in the ABINIT material science package [17]. The implementation, realized in fortran90, benefits from utilizing the generic linear algebra routines available in the CUDA [37] and MAGMA [22][39] GPU libraries. More

recently, NVIDIA announced that LOBPCG will be included in the GPU-accelerated Algebraic Multigrid Accelerator AmgX ¹.

LOBPCG

LOBPCG stands for Locally Optimal Block Preconditioned Conjugate Gradient method [27][26]. It is designed to find m of the smallest (or largest) eigenvalues λ and corresponding eigenvectors x of a symmetric and positive definite eigenvalue problem:

$$Ax = \lambda x.$$

Similarly to other CG-based methods, this is accomplished by the iterative minimization of the Rayleigh quotient:

$$\rho(x) = \frac{x^T Ax}{x^T x},$$

which results in finding the smallest eigenstates of the original problem. In the LOBPCG method the minimization at each step is done locally, in the subspace of the current approximation x_i , the previous approximation x_{i-1} , and the preconditioned residual $P(Ax_i - \lambda_i x_i)$, where P is a preconditioner for A . The subspace minimization is done by the Rayleigh-Ritz method.

Note that the operations in the algorithm are blocked and therefore can be very efficient on modern architectures. Indeed, the AX_i is the SpMM kernel, and the bulk of the computations in the Rayleigh-Ritz minimization are general matrix-matrix products (GEMMs). The direct implementation of this algorithm becomes unstable as the difference between X_{i-1} and X_i becomes small, and therefore special care and modifications must be taken (see [27][21]). While the LOBPCG convergence characteristics usually benefit from using an application-specific preconditioner [7][12][29][30][25], we refrain from including preconditioners as we are particularly interested in the performance of the top-level method. Our implementation is hybrid, using both the GPUs and CPUs available. In particular, all data resides on the GPU memory and the bulk of the computation – the preconditioned residual, the accumulation of the matrices for the Rayleigh-Ritz method, and the update transformations – are done on the GPU. The small and not easy to parallelize Rayleigh-Ritz eigenproblem is done on the CPU using vendor-optimized LAPACK. For stability, various orthogonalizations are performed, following the LOBPCG Matlab code from A. Knyazev ². We used our highly optimized GPU implementations based on the Cholesky QR to get the same convergence rates as the reference CPU implementation from BLOPEX (in HYPRE) on all our test matrices from the University of Florida sparse matrix collection (see Section). More stable versions, including Cholesky/SVD QR iterations and the mixed-precision

¹<https://developer.nvidia.com/amgx>

²<http://www.mathworks.com/matlabcentral/fileexchange/48-lobpcg-m>

Cholesky QR [43], as well as LAPACK/MAGMA based, CGS, and MGS for GPUs are also an option that we provide.

SPMM PRODUCT

A key building block for the LOBPCG algorithm and other block-Krylov solvers is a routine generating the Krylov search directions by computing the product of a sparse matrix and a set of vectors. This routine can obviously be implemented as a set of consecutive sparse matrix vector products; however, the interpretation as a product of a sparse matrix and a tall-and-skinny dense matrix composed of the distinct vectors may promote a different approach (sparse matrix dense matrix product, SpMM). In particular, already cached data of the sparse matrix may be reused when processing multiple vectors simultaneously. This would render performance improvement to the memory-bound kernel. In the GPU implementation of LOBPCG, we realize this routine by handling the sparse matrix using the recently proposed SELL-P format (padded sliced ELLPACK format [6]). In the following we first describe the SELL-P format, provide details on how we implement the SpMM kernel, and then analyze its performance by comparing against the CSRSpMM taken from NVIDIA’s CUSPARSE library [35].

Implementation of SpMM for SELL-P

While for dense matrices it is usually reasonable to store all matrix entries in consecutive order, sparse matrices are characterized by a large number of zero elements, and storing those is not only unnecessary, but would also incur significant storage overhead. Different storage layouts exist that aim to reducing the memory footprint of the sparse matrix by storing only a fraction of the elements explicitly, and anticipating all other elements to be zero, see [10][40134013]. In the CSR format [10], this idea is taken to extremes, as only nonzero entries of the matrix are stored. In addition to the array **values** containing the nonzero elements, two integer arrays **colind** and **rowptr** are used to locate the elements in the matrix. While this storage format is suitable when computing a sparse matrix vector product on processors with a deep cache-hierarchy, as it reduces the memory requirements to a minimum, it fails to allow for high parallelism and coalesced memory access when computing on streaming-processors like GPUs. On those, the ELLPACK-format, padding the different rows with zeros for a uniform row-length, coalesced memory access, and instruction parallelism may, depending on the matrix characteristics, outperform the CSR format [11]. However, the ELLPACK format incurs a storage overhead for the general case: The maximum number of nonzero elements aggregated in one row determines how many elements are stored per row – eventually filled with explicit zeros. Hence, the overhead is determined by the maximum number of nonzeros in one row and the average number of nonzeros per row (see Table 1). Depending on the associated memory and computational

overheads, using ELLPACK may result in poor performance, despite that coalesced memory access is highly favourable for streaming processors.

A workaround to reduce memory and computational overhead is to split the original matrix into row blocks before converting these into the ELLPACK format. In the resulting sliced ELLPACK format (SELL or SELL-C where C denotes the size of the row blocks [32][31]), the overhead is no longer determined by the matrix row containing the largest number of nonzeros, but by the row with the largest number of nonzero elements in the respective block. While sliced SELL-C reduces the overhead very efficiently (i.e., choosing C=1 results in the storage-optimal CSR format), assigning multiple threads to each row requires padding the rows with zeros, such that each block has a rowlength divisible by this thread number. This is the underlying idea of the SELL-P format: partition the sparse matrix into row-blocks, and convert the distinct blocks into ELLPACK format [11] with the rowlength of each block being padded a multiple of the number of threads assigned to each row when computing a matrix vector or matrix multi-vector product.

Although the padding introduces some zero fill-in, the comparison between the formats in Table 1 reveals that the blocking strategy may still render significant memory savings compared to ELLPACK, which directly translate into reduced computational cost for the SpMV kernel. For the design of the SpMM routine it is not sufficient to reduce the computational overhead, as performance also depends on the memory bandwidth. Therefore, it is essential to optimize the memory access pattern, which requires the accessed data to be aligned in memory whenever possible [37]. For consecutive memory access, and with the motivation of processing multiple vectors simultaneously, we implement the SpMM assuming the tall-and-skinny dense matrix composed of the vectors being stored in row-major order. Although this requires a pre-processing step transposing the dense matrix prior to the SpMM call, the more appealing aligned memory access to the vector values may compensate for the extra work.

The SpMM kernel then arises as a natural extension of the SpMV routine for the SELL-P format proposed in [6]. Like in the SpMV kernel, the x-dimension of the thread block processes the distinct rows of one SELL-P block, while the y-dimension corresponds to the number of threads assigned to each row, see Figure 1. Partial products are written into shared memory and added in a local reduction phase. For the SpMM it is beneficial to process multiple vectors simultaneously, which motivates for extending the thread block by a z-dimension, handling the distinct vectors. While assigning every z-layer of the block to one vector would provide a straight-forward implementation, keeping the set of vectors (respectively the tall-and-skinny dense matrix), in texture memory, makes an enhanced approach more appealing. The motivation is that in CUDA (version 5.5) every texture read

Acronym	Matrix	#nonzeros (n_z)	Size (n)	n_z/n	n_z^{row}	ELLPACK		SELL-P	
						$n_z^{ELLPACK}$	overhead	n_z^{SELL-P}	overhead
AUDI	AUDIKW_1	77,651,847	943,645	82.28	345	325,574,775	76.15%	95,556,416	18.74%
BMW	BMWCRA1	10,641,602	148,770	71.53	351	52,218,270	79.62%	12,232,960	13.01%
BONE010	BONE010	47,851,783	986,703	48.50	64	62,162,289	23.02%	55,263,680	13.41%
CRANK	CRANKSEG_2	14,148,858	63,838	221.63	3423	218,517,474	93.53%	15,991,232	11.52%
F1	F1	26,837,113	343,791	78.06	435	149,549,085	82.05%	33,286,592	19.38%
INLINE	INLINE_1	38,816,170	503,712	77.06	843	424,629,216	91.33%	45,603,264	19.27%
LDOOR	LDOOR	42,493,817	952,203	44.62	77	73,319,631	42.04%	52,696,384	19.36%

Table 1: Matrix characteristics and storage overhead for selected test matrices from the Tim Davis Matrix Collection [15] when using ELLPACK, or SELL-P format. SELL-P employs a blocksize of 8 with 4 threads assigned to each row. n_z^{FORMAT} refers to the explicitly stored elements (n_z nonzero elements plus the explicitly stored zeros for padding).

fetches 16 bytes, corresponding to two IEEE double or four IEEE single precision floating point values. As using only part of them would result in performance waste, every z-layer may process two (double precision case) or four (single precision case) vectors, respectively. This implies that, depending on the precision format, the z-dimension of the thread block equals half or a quarter the column count of the tall-and-skinny dense matrix.

As assigning multiple threads to each row requires a local reduction of the partial products in shared memory (see Figure 1), the x- y- and z- dimensions are bounded by the characteristics of the GPU architecture [37]. An efficient workaround when processing a large number of vectors is given by assigning only one thread per z-dimension to each row (choose y-dimension equal 1), which removes the reduction step and the need for shared memory.

EXPERIMENTAL RESULTS

Hardware Setup: We use two Intel Xeon E5-2670 (Sandy Bridge) CPUs accelerated by an NVIDIA Tesla K40c GPU with a theoretical peak performance of 1,682GFLOP/s. The host system has a theoretical peak of 333GFLOP/s, main memory size is 64 GB, and theoretical bandwidth is up to 51 GB/s. On the K40 GPU, 12 GB of main memory are accessed at a theoretical bandwidth of 288 GB/s. The implementation of all GPU kernels is realized in CUDA [37], version 5.5 [36], while we also include in the performance comparisons routines taken from NVIDIA’s cuSPARSE [35] library. On the CPU, Intel’s MKL [1] is used in version 11.0, update 5. Note that the CPU-based implementations use the “numactl -interleave=all” option when beneficial.

Performance of SpMM for SELL-P: In Table 2 we compare the asymptotic performance achieved by the SpMV and SpMM kernels taken from Intel’s MKL library [1] and NVIDIA’s cuSPARSE library [35] with the developed SELL-P SpMV and SpMM kernels that are available in the MAGMA open source software stack [22].

Both GPU implementations for the SpMM (cuSPARSE and MAGMA) assume the vectors to be stored in row-major data format. This typically results in significantly higher performance [34]. As most algorithms require column-major storage, and for a fair comparison to the CPU implementation, the transposition operation is included in the performance of those kernels. The CPU

Matrix	Intel MKL		NVIDIA cuSPARSE			MAGMA	
	CSR	SpMM	CSR	HYB	SpMM	SELL-P	SpMM
AUDI	7.24	22.5	21.9	19.3	88.95	22.1	104.21
BMW	6.86	32.2	17.7	16.9	93.04	23.6	112.46
BONE010	7.77	30.5	22.3	20.7	87.71	22.3	108.26
F1	5.64	20.1	24.2	19.1	82.15	19.6	99.63
INLINE	8.10	28.9	15.5	14.9	87.76	21.1	102.00
LDOOR	6.78	41.5	25.2	19.3	83.09	20.7	99.37

Table 2: Asymptotic DP performance [GFLOP/s] of sparse test matrices and a large number of vectors with a set of consecutive SpMVs (MKL CSR, cuSPARSE CSR, cuSPARSE HYB, MAGMA SELL-P SpMV) and the respective SpMM kernels. See Table 1 for the respective matrix characteristics.

runs are using the numactl --interleave=all policy, which is well known to improve performance. The performance obtained for the MKL routines is consistent with benchmarks provided by Intel [2].

Depending on the matrix characteristics, the GPU SpMVs are between 2 and 4× faster than MKL. This is expected from the compute and bandwidth capabilities of the two architectures. Comparing the performance of the GPU kernels, the SELL-P kernel is the winner in 4 or 6 cases. On the CPU, the MKL SpMM routine achieves about 4.1× better performance than the SpMV kernel. Similar improvements can be observed on the GPU: The performance of the cuSPARSE SpMM kernel is 4.1× higher than the CSR kernel, and 4.7× higher than the HYB kernel; The MAGMA SpMM kernel achieves 4.8× better performance than the SpMV kernel. Comparing CPU with GPU, the cuSPARSE SpMM is on average 3× faster than the MKL SpMM, however outperformed for all test matrices by the developed SELL-P SpMM kernel, accelerating the computation by factors between 2.5 and 5. With an average speedup of 3.6 over the MKL, the SELL-P SpMM performance typically exceeds 100 GFLOP/s.

LOBPCG GPU Performance: Finally, we want to quantify how the developed SpMM improves the performance of the LOBPCG GPU implementation. For this purpose, we benchmark two versions of the LOBPCG implementation, one using a set of consecutive SpMVs to generate the search directions, and one where we integrate the developed SpMM kernel. Furthermore, we compare against the multithreaded CPU implementation of LOBPCG provided by Andrew Knyazev in the BLOPEX package [24]. As popular software libraries

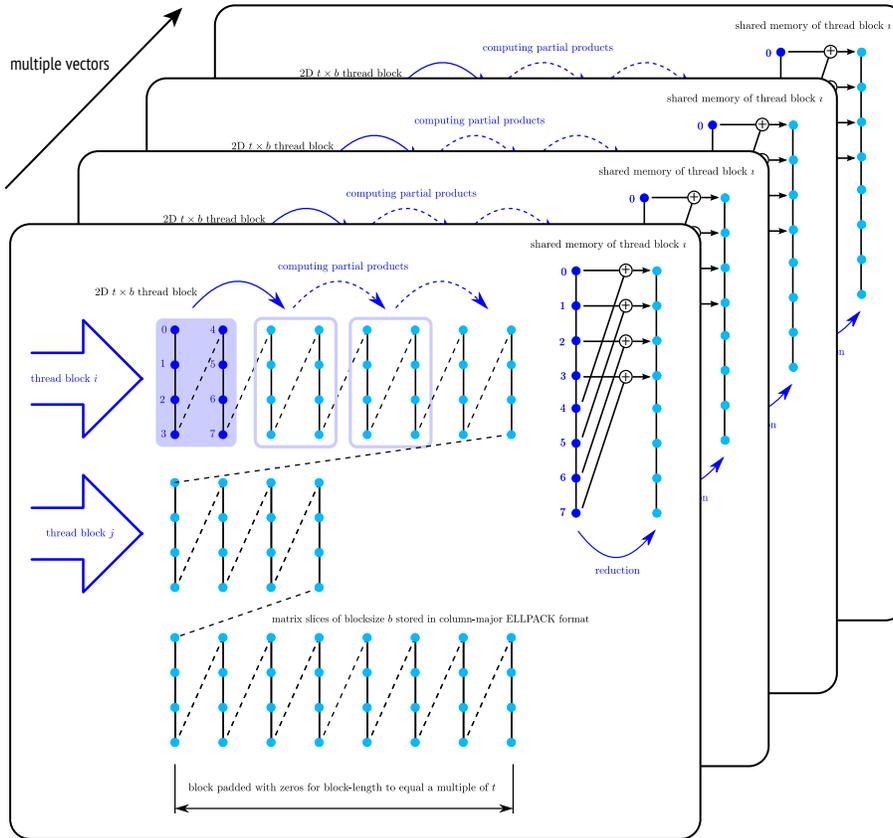


Figure 1: SELL-P memory layout and SpMM kernel including the reduction step using blocksize $b = 4$ (SELL-4), and assignment of two threads to row ($t = 2$). Adding a z-dimension to the thread-block allows to process multiple vectors simultaneously.

like PETSc [9] and Hypre [18] provide interfaces to this implementation [28], we may consider this code as the state-of-the-art CPU implementation of LOBPCG. For the benchmark results, we used the BLOPEX code via its the Hypre interface. For optimal utilization of the Sandy Bridge architecture, we enable hyperthreading and execute the eigensolver using 32 OpenMP threads.

The LOBPCG implementation in BLOPEX is matrix free, i.e., the user is allowed to provide their choice of SpMV/SpMM implementation. In these experiments we use the Hypre interface to BLOPEX, linked with the MKL library.

The convergence on the GPU is matching the BLOPEX convergence. The number of operations executed in every iteration of LOBPCG can be approximated by

$$2 \cdot nnz \cdot n_v + 36 \cdot n \cdot n_v^2 \quad (1)$$

where nnz denotes the number of nonzeros of the sparse matrix, n the dimension and n_v the number of eigenvectors (equivalent to the number of columns in the tall-and-skinny dense matrix). The left part of the sum reflects the SpMM operation generating the Krylov vectors, the right part contains the remaining operations including the orthogonalization of the search directions.

Due to the n_v^2 term, we may expect the runtime to increase superlinearly with the number of vectors, which can be observed in Figure 2 where we visualize the time needed to complete 100 iterations on the AUDI problem using either the BLOPEX code via the Hypre interface or the GPU implementation using either a sequence of SpMVs or the SpMM kernel to generate the search directions. Comparing the results for the AUDI problem, we are 1.3 and $1.2\times$ faster when computing 32 and 48 eigenvectors, respectively, using the SpMM instead of the SpMV in the GPU implementation of LOBPCG. Note that although in this case the SpMM performance is about $5\times$ the SpMV performance, the overall improvement of correspondingly 30% and 20% reflects that only 12.5% and 8.7% of the overall LOBPCG flops are in SpMVs for the 32 and 48 eigenvector problems, respectively (see equation (1) and the matrix specifications in Table 1). While the BLOPEX implementation also shows some variances for different numbers of vectors, the runtime pattern of the GPU LOBPCG reflects the efficiency of the orthogonalization routines favoring cases where 16, 32, or 48 vectors are processed. This pattern is amplified when replacing the consecutive SpMVs with the SpMM, as this kernel also promotes certain column-counts of the tall and skinny dense matrix.

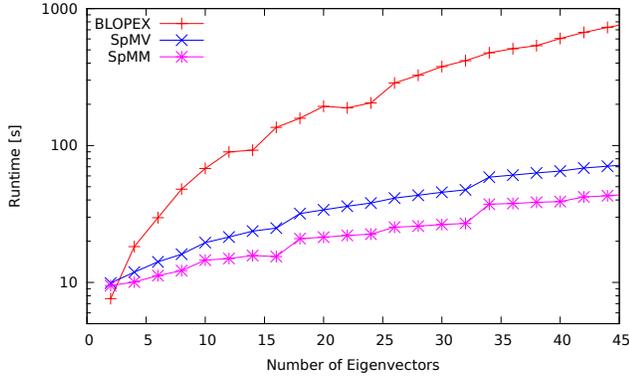


Figure 2: Runtime to complete 100 iterations on the AUDI problem using either the BLOPEX code via the Hypre interface or the GPU implementation using either a sequence of SpMVs or the SpMM kernel to generate the search directions.

To complete the performance analysis, we report in Figure 3 the speedup factors of the GPU LOBPCG *vs.* the BLOPEX code via its Hypre interface. We observe that as soon as 16 eigenvectors are needed, the GPU implementation using the consecutive SpMVs outperforms the CPU by more than 5 \times , while for the SpMM-based algorithm the acceleration is on average close to 8 \times . The 5 \times speedup when using the consecutive SpMVs on the GPU indicates that the Hypre interface to LOBPCG is not blocking the SpMVs. Based on the kernels' analysis, the expectation is that an optimized CPU code (blocking the SpMVs) would achieve about the same performance as the GPU LOBPCG without blocking, which is about 3 to 5 \times slower than the blocked version. Computing more vectors reduces the fraction of SpMV flops to the total flops (see equation (1)), and thus making the SpMV implementation less critical for the overall performance. The fact that the speedup of the GPU *vs.* the CPU LOBPCG continues to grow, reaching 20 and up to 35 \times for 48 vectors, shows that there are other missed optimization opportunities in the CPU implementation. In particular, these are the GEMMs in assembling the matrix representations for the local Rayleigh-Ritz minimizations, and the orthogonalizations. These routines are highly optimized in our GPU implementation, especially the GEMMs, which due to the specific sizes of the matrices involved – tall and skinny matrices A and B with a small square resulting matrices $A^T B$ – required modifications to the standard GEMM algorithm for large matrices [33]. Benefits are in particular drawn from splitting the $A^T B$ GEMM into smaller GEMMs based on tuning the MAGMA GEMM [33] for small sizes. The execution is then grouped into a single batched GEMM, followed by addition of the local results [43]. Based on our performance analysis, the acceleration factor against a similarly optimized CPU code would be in the range of 2.5 to 5 \times .

SUMMARY AND OUTLOOK

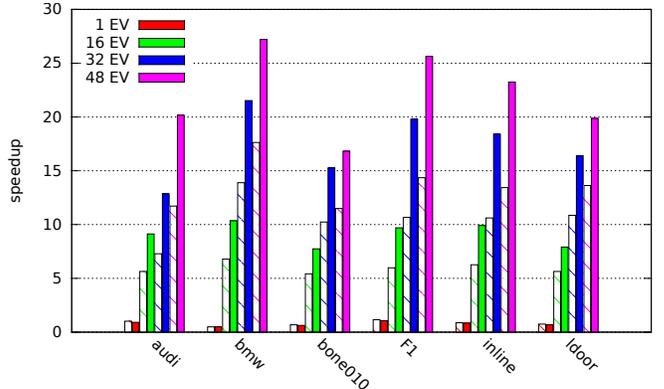


Figure 3: Speedup of the GPU LOBPCG over CPU BLOPEX (Hypre interface linked with MKL) using consecutive SpMVs (striped) and the SpMM kernel (solid), respectively.

We presented a heterogeneous CPU-GPU algorithm design for the LOBPCG eigensolver. The benefit of using blocking routines like the LOBPCG is based on a more efficient use of hardware. As opposed to running at the low performance of a SpMV kernel, which is typical for Krylov subspace methods, the presented LOBPCG runs at the speed of a SpMM kernel designed for the SELL-P format. On NVIDIA's K40 GPU, this kernel outperforms Intel's SpMM on two eight-core Intel Sandy Bridge cores by 2.5 to 5 \times . Instead of the standard Krylov subspace methods' memory-bound performance of 20 to 25 GFlop/s (in double precision on a K40), the LOBPCG computes a small set of eigenstates at a typical rate of 100 to 140 GFlop/s. Our heterogeneous LOBPCG outperformed the Hypre interface of the BLOPEX CPU implementation by more than an order of magnitude when computing a small set of eigenstates. This reveals that even for multicore CPUs, where the HPC software stack is considered to be better established than the HPC software stack for the more recent GPU architectures, there are many missed optimization opportunities. The developed SpMM routine, the specific GEMMs, and orthogonalizations, serve as key building blocks not only block-Krylov, but also for other methods that rely on blocking strategies. Hence, the kernel design and findings presented in this paper may be used to accelerate other methods in a similar fashion.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. ACI-1339822, Department of Energy grant No. DE-SC0010042, NVIDIA, and the Russian Scientific Fund (Agreement N14-11-00190).

REFERENCES

1. Intel® Math Kernel Library for Linux* OS. Document Number: 314774-005US, October 2007. Intel.
2. Intel® Math Kernel Library. Sparse BLAS and Sparse Solver Performance Charts: DCSRGMV and DCSRMM, October 2014. Intel Corporation.
3. Agullo, E., Augonnet, C., Dongarra, J., Faverge, M., Ltaief, H., Thibault, S., and Tomov, S. QR Factorization on a Multicore Node Enhanced with Multiple GPU Accelerators. In *IPDPS*, IEEE (2011), 932–943.
4. Anderson, E., Bai, Z., Bischof, C., Blackford, L. S., Demmel, J., Dongarra, J. J., Du Croz, J., Hammarling, S., Greenbaum, A., McKenney, A., and Sorensen, D. *LAPACK Users' Guide (Third Ed.)*. SIAM, Philadelphia, PA, USA, 1999.
5. Anderson, M., Ballard, G., Demmel, J., and Keutzer, K. Communication-avoiding qr decomposition for gpus. Tech. Rep. UCB/EECS-2010-131, EECS Department, UC Berkeley, Oct 2010.
6. Anzt, H., Tomov, S., and Dongarra, J. Implementing a Sparse Matrix Vector Product for the SELL-C/SELL-C- σ formats on NVIDIA GPUs. Tech. Rep. ut-eecs-14-727, University of Tennessee, March 2014.
7. Arbenz, P., and Geus, R. Multilevel preconditioned iterative eigensolvers for maxwell eigenvalue problems. *Applied Numerical Mathematics* 54, 2 (2005), 107 – 121. 6th IMACS International Symposium on Iterative Methods in Scientific Computing.
8. Baker, C. G., Hetmaniuk, U. L., Lehoucq, R. B., and Thornquist, H. K. Anasazi software for the numerical solution of large-scale eigenvalue problems. *ACM Trans. Math. Softw.* 36, 3 (July 2009), 13:1–13:23.
9. Balay, S., Adams, M. F., Brown, J., Brune, P., Buschelman, K., Eijkhout, V., Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L. C., Rupp, K., Smith, B. F., and Zhang, H. PETSc Web page. <http://www.mcs.anl.gov/petsc>, 2014.
10. Barrett, R., Berry, M., Chan, T. F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., and der Vorst, H. V. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.
11. Bell, N., and Garland, M. Efficient sparse matrix-vector multiplication on CUDA, Dec. 2008.
12. Benner, P., and Mach, T. Locally optimal block preconditioned conjugate gradient method for hierarchical matrices. *PAMM* 11, 1 (2011), 741–742.
13. Buluç, A., Williams, S., Olike, L., and Demmel, J. Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication. In *Proc. IPDPS* (2011), 721–733.
14. Castro, A., Appel, H., Oliveira, M., Rozzi, C. A., Andrade, X., Lorenzen, F., Marques, M. A. L., Gross, E. K. U., and Rubio, A. octopus: a tool for the application of time-dependent density functional theory. *phys. stat. sol. (b)* 243, 11 (2006), 2465–2488.
15. Davis, T. A., and Hu, Y. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.* 38, 1 (Dec. 2011), 1:1–1:25.
16. Demmel, J., Grigori, L., Hoemmen, M., and Langou, J. Communication-avoiding parallel and sequential qr factorizations. *CoRR abs/0806.2159* (2008).
17. et al., X. G. First-principles computation of material properties: the ABINIT software project. *Computational Materials Science* 25, 3 (2002), 478 – 492.
18. Falgout, R., and Yang, U. Hypre: A Library of High Performance Preconditioners. In *ICCS 2002, Amsterdam, The Netherlands, April 21-24, 2002. Proceedings, Part III*, P. M. A. Sloot, C. J. K. Tan, J. Dongarra, and A. G. Hoekstra, Eds., vol. 2331 of *Lecture Notes in Computer Science*, Springer (2002), 632–641.
19. Golub, G. H., and Van Loan, C. F. *Matrix Computations (3rd Ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.
20. Heroux, M., Bartlett, R., Hoekstra, V. H. R., Hu, J., Kolda, T., Lehoucq, R., Long, K., Pawlowski, R., Phipps, E., Salinger, A., Thornquist, H., Tuminaro, R., Willenbring, J., and Williams, A. An Overview of Trilinos. Tech. Rep. SAND2003-2927, Sandia National Laboratories, 2003.
21. Hetmaniuk, U., and Lehoucq, R. Basis selection in LOBPCG. *Journal of Computational Physics* 218, 1 (2006), 324 – 332.
22. Innovative Computing Laboratory, UTK. Software distribution of MAGMA version 1.5. <http://icl.cs.utk.edu/magma/>, 2014.
23. Jones, E., Oliphant, T., Peterson, P., et al. SciPy: Open source scientific tools for Python, 2001–.
24. Knyazev, A. <https://code.google.com/p/blkpex/>.
25. Knyazev, A., and Neymeyr, K. *Efficient Solution of Symmetric Eigenvalue Problems Using Multigrid Preconditioners in the Locally Optimal Block Conjugate Gradient Method*. UCD/CCM report. University of Colorado at Denver, 2001.
26. Knyazev, A. V. Preconditioned eigensolvers - an oxymoron? *Electronic Transactions on Numerical Analysis* 7 (1998), 104–123.

27. Knyazev, A. V. Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method. *SIAM J. Sci. Comput* 23 (2001), 517–541.
28. Knyazev, A. V., Argentati, M. E., Lashuk, I., and Ovtchinnikov, E. E. Block locally optimal preconditioned eigenvalue solvers (blopex) in hypre and petsc. *SIAM J. Scientific Computing* 29, 5 (2007), 2224–2239.
29. Kolev, T. V., and Vassilevski, P. S. Parallel eigensolver for H(curl) problems using H1-auxiliary space AMG preconditioning. Tech. Rep. UCRL-TR-226197, Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2006.
30. Kressner, D., Pandur, M. M., and Shao, M. An indefinite variant of lobpcg for definite matrix pencils. *Numerical Algorithms* (2013), 1–23.
31. Kreutzer, M., Hager, G., Wellein, G., Fehske, H., and Bishop, A. R. A unified sparse matrix data format for modern processors with wide simd units. *CoRR abs/1307.6209* (2013).
32. Monakov, A., Lokhmotov, A., and Avetisyan, A. Automatically tuning sparse matrix-vector multiplication for gpu architectures. In *Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers, HiPEAC'10*, Springer-Verlag (Berlin, Heidelberg, 2010), 111–125.
33. Nath, R., Tomov, S., and Dongarra, J. An improved magma gemm for fermi graphics processing units. *Int. J. High Perform. Comput. Appl.* 24, 4 (Nov. 2010), 511–515.
34. Naumov, M. Preconditioned block-iterative methods on gpus. *PAMM* 12, 1 (2012), 11–14.
35. NV. *CUSPARSE LIBRARY*, July 2013.
36. NVIDIA. *NVIDIA CUDA TOOLKIT V5.5*, July 2013.
37. NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*, 2.3.1 ed., August 2009.
38. Saad, Y. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003.
39. Tomov, S., Dongarra, J., and Baboulin, M. Towards dense linear algebra for hybrid gpu accelerated manycore systems. *Parallel Comput. Syst. Appl.* 36, 5-6 (2010), 232–240. <http://dx.doi.org/10.1016/j.parco.2009.12.005> DOI: 10.1016/j.parco.2009.12.005.
40. Williams, S., Bell, N., Choi, J., Garland, M., Oliker, L., and Vuduc, R. Sparse matrix vector multiplication on multicore and accelerator systems. In *Scientific Computing with Multicore Processors and Accelerators*, J. Kurzak, D. A. Bader, and J. Dongarra, Eds. CRC Press, 2010.
41. Yamada, S., Imamura, T., Kano, T., and Machida, M. High-performance computing for exact numerical approaches to quantum many-body problems on the earth simulator. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, SC '06*, ACM (New York, NY, USA, 2006).
42. Yamada, S., Imamura, T., and Machida, M. 16.447 tflops and 159-billion-dimensional exact-diagonalization for trapped fermion-hubbard model on the earth simulator. SC '05, IEEE Computer Society (Washington, DC, USA, 2005), 44–.
43. Yamazaki, I., Tomov, S., Dong, T., and Dongarra, J. Mixed-precision orthogonalization scheme and adaptive step size for CA-GMRES on GPUs. *VECPAR* (jan 2014).

Incremental, Distributed Single-Linkage Hierarchical Clustering Algorithm Using MapReduce

Chen Jin, Zhengzhang Chen, William Hendrix, Ankit Agrawal, Alok Choudhary
Northwestern University, Evanston, IL
{chen.jin, zzc472, whendrix, ankitag, choudhar}@eecs.northwestern.edu

Author Keywords

Hierarchical Clustering, Incremental Hierarchical Clustering, MapReduce

ABSTRACT

Single-linkage hierarchical clustering is one of the prominent and widely-used data mining techniques for its informative representation of clustering results. However, the parallelization of this algorithm is challenging as it exhibits inherent data dependency during the hierarchical tree construction. Moreover, in many modern applications, new data is continuously added into the already huge datasets. It would be impractical to reapply the clustering algorithm on the augmented datasets from scratch.

In this paper, we propose a unified algorithm which can not only cluster the large dataset, but also incorporate the newly arrived data incrementally. More specifically, we formulate the single-linkage hierarchical clustering problem as a Minimum Spanning Tree (MST) construction problem on a complete graph. The algorithm decomposes the complete graph into a set of non-overlapped subgraphs, computes the intermediate sub-MSTs for each subgraph in parallel, and merges all the sub-MSTs to achieve the final solution. In addition, the same framework can treat the incremental data insertion as a separate data subset and integrate it nicely with the existing solution. We implement the unified algorithm by employing MapReduce framework.

Using both synthetic and real-world datasets containing up to millions of high-dimensional points, we show that the proposed algorithm achieves a scalable speedup up to 200 on 300 computer cores for the base dataset and a speedup up to 120 for the dataset with maximum 5% random insertion.

INTRODUCTION

Hierarchical clustering is one of the prominent and widely-used data mining techniques for its informative representation of clustering results. It organizes the relationships of clusters using a tree diagram (dendrogram), giving the idea of how each data point is positioned related to the overall cluster structure. Compared to non-parameter free clustering algorithms like partitioning clustering, hierarchical clustering does not require the number of clusters in advance, and can deterministically assign the cluster label to each data point. Moreover, it offers insight into the complete hierarchy of clusters. As a representative implementation of hierarchical clustering algorithm, single-linkage method has been used in

numerous applications such as document classification, computational biology, and image segmentation [19, 22, 23, 28]. For example, [5] uses dendrogram to visualize hierarchical clustering of tissues and genes; [28] employs hierarchical information to help visitors navigate the articles on the web-sites.

In the face of the ever-growing datasets, the single-machine performance of hierarchical clustering algorithm can no longer keep up the game, which creates an urgent demand for a parallel solution. However, the parallelization of hierarchical clustering algorithm is a non-trivial task. First, hierarchical clustering algorithm itself is highly computationally expensive. And, during the hierarchical tree construction procedure, it exhibits inherent data dependency. Second, in many recent applications, data are often received incrementally and merged to already huge datasets, it would be impractical to reapply the clustering algorithm on the updated dataset from the scratch whenever new data arrives. For instance, the data warehouses collect new data and apply updates periodically in a batch mode at the off-peak time; Facebook has more than 100 million photos uploaded per day and 90 billion photos in total currently [1]. It is impractical or even infeasible to apply the clustering algorithm on the entire dataset. Thus, efficiently updating the clustering solution incrementally on the explosive size of the original dataset without applying the algorithm from scratch is challenging but critical.

To cope with these two challenges gracefully without designing two different frameworks, a unified algorithm is required to not only cluster the large datasets efficiently, but also incorporate the newly added data incrementally. In this paper, we propose IncDiSC, an Incremental, Distributed Single-linkage hierarchical Clustering algorithm using MapReduce framework. The idea of our algorithm is to formulate the single-linkage hierarchical clustering algorithm using graph algorithmic concepts, which is equivalent to solve a Minimum Spanning Tree of the complete graph induced by the dataset [17]. The algorithm initially divides the base dataset into a number of non-overlapped subsets. Therefore, the complete graph induced by the base dataset can be decomposed into a set of complete subgraphs induced by data subset respectively as well a number of complete bipartite subgraphs by pairs of data subsets. In this way, any edge in the original complete graph is assigned to only one subgraph and all subgraphs are independent from one another. Such a divide-and-conquer strategy allows us to solve sub-MSTs in parallel and merge them to achieve the final MST.

When there is a new dataset come in, IncDiSC forms the complete bipartite subgraphs between the new dataset and each

initial data subset, as well as the complete subgraph induced solely by the new dataset. In the same vein, the algorithm calculates sub-MSTs for all the subgraphs and merge them with the existing MST to find the MST for the augmented dataset. This algorithm can be applied periodically at the data warehouse with relatively low cost. The algorithm provides both incremental and scalable solution to handle large-scale dynamic datasets. It is memory-efficient and can be scaled out linearly. In the experiment section, we present a data-dependent characterization of hardness and evaluate algorithm’s scalability with up to 1 million data points. The k-way merge process of intermediate solutions is configurable by setting user-defined partitioning function in MapReduce framework. The empirical results show our algorithm can achieve an estimated speedup up to 200 on 300 computer cores for the base dataset and a speedup up to 120 for the dataset with maximum 5% random insertion.

The main contributions of this paper are summarized as follows:

- We propose a scalable algorithm for single-linkage hierarchical clustering using MapReduce framework which also support an incremental update on the original solution;
- We introduce a configurable merge process achieving reasonable trade-off between the number of MapReduce rounds and the degree of parallelism;
- And empirical evaluation demonstrates the linear scalability and speedup on both synthetic and real-world data.

The remainder of paper is organized as follows. We brief the related work in Section II. In Section III, we describe our proposed incremental, distributed single-linkage hierarchical clustering algorithm, and we detail the system design using MapReduce framework. In Section IV, the algorithm’s scalability is evaluated on both synthetic datasets and real-world datasets. Finally, we draw the conclusions in Section V.

RELATED WORK

The goal of this paper is to develop a new parallel, incremental hierarchical clustering algorithm using MapReduce. Thus, our work is related to parallelization of hierarchical clustering. Since our approach essentially reformulates the hierarchical clustering problem as finding a Minimum Spanning Tree in the complete connected graph induced by the dataset, we have investigated literature on solving MST problems using MapReduce. On the other hand, some researchers have proposed incremental variations for hierarchical clustering algorithms in the context of sequential implementation. In the following, we discuss most recent research results from these perspectives.

PARALLELIZATION OF HIERARCHICAL ALGORITHM: In order to overcome the inefficiency of the sequential hierarchical clustering algorithm on large-scale, high dimensional dataset, Hendrix *et al.* present SHRINK [17], a parallel single-linkage hierarchical clustering algorithm based on SLINK [3]. SHRINK exhibits good scaling and communication behavior, and only keeps space complexity in $\mathcal{O}(n)$ with n being the number of data points. The algorithm trades

duplicated computation for the independence of the subproblem, and leads to a good speedup. However, this work only evaluates SHRINK on up to 36 shared memory cores, achieving a speedup of roughly 19.

As a powerful data processing tool, MapReduce is gaining significant momentum from both industry and academia. Some researchers have started to explore the possibility of implementing hierarchical clustering algorithm using MapReduce framework. For example, Wang and Dutta present PARABLE [29], a parallel hierarchical clustering algorithm using MapReduce. The algorithm is decomposed into two stages. In the first stage, the mappers randomly split the entire dataset into smaller partitions, on each of which the reducers perform the sequential hierarchical clustering algorithm. The intermediate dendrograms from all the small partitions are aligned and merged into a single dendrogram to suggest a final solution. However, this work does not provide the formal theoretical proof on the correctness of the dendrogram alignment algorithm. And the experiments only use 30 mappers and 30 reducers for the local clustering and a single reducer for the final global clustering. Hierarchical clustering has also been parallelized using other frameworks such as MPI [9] and GPU [10]. However, most of these methods explicitly compute and store the entire distance matrix containing all the pair-wise distance of the dataset.

SOLVING MST USING MAPREDUCE: Recently, Rastogi *et al.* [26] propose an efficient algorithm to find all the connected components in logarithmic number of MapReduce iterations for large-scale graphs. They present four different hashing schemes, among which Hash-to-Min proved to finish in $\mathcal{O}(\log n)$ iterations for path graphs and $\mathcal{O}(k(|V| + |E|))$ communication cost at round k . In the same paper, the algorithm is applied to single-linkage hierarchical clustering. It starts with each vertex and its neighbors as a starting connected component, all the components hashed to the same reducer are merged to a bigger component; a MST algorithm is then applied. However, a separate MapReduce job is required to determine the stop condition at each iteration, which might be acient. Besides, different from this work on general graphs, our work focuses on on complete connected graphs.

Lattanzi *et al.* [21] present Filtering, a novel method for solving large-scale dense graph problem in MapReduce. The authors present algorithms for MSTs as well as other fundamental graph problems with a constant number of MapReduce rounds even with machines having substantially sub-linear memory. However, the algorithms are mainly focused on meeting memory constraints rather than improving the scalability, and they only provide theoretical results.

SEQUENTIAL INCREMENTAL HIERARCHICAL CLUSTERING: Chen *et al.* [11] propose the incremental hierarchical clustering algorithm GRIN for numerical datasets. The algorithm is conducted in two phases in order to build the updated dendrogram. In the first phase, GRACE, a gravity-based agglomerative hierarchical clustering algorithm is applied to build a clustering dendrogram for the sets. Then the clustering dendrogram is reconstructed by flatterring and pruning its bottom levels to generate a tentative dendrogram before

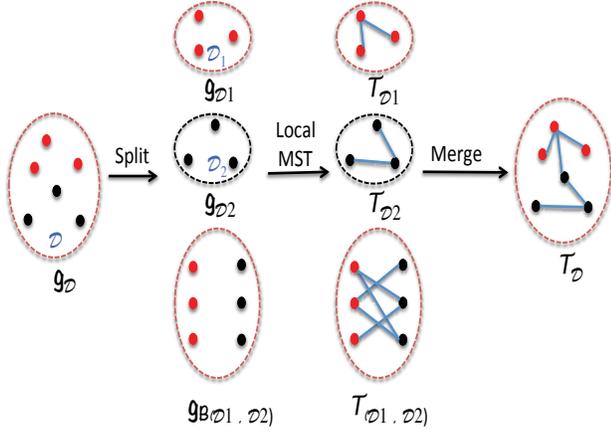


Figure 1: Illustration of parallel strategy on base dataset \mathcal{D} . We divide dataset \mathcal{D} into two smaller parts, \mathcal{D}_1 and \mathcal{D}_2 , calculate MSTs for \mathcal{D}_1 , \mathcal{D}_2 , and the complete bipartite graph between them, then merge the MSTs to find the final MST for \mathcal{D} .

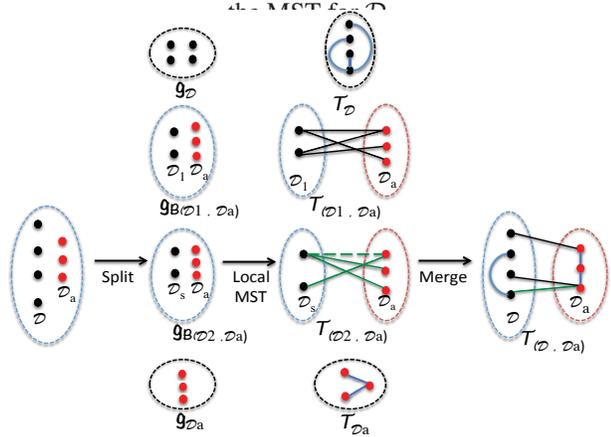


Figure 2: Illustration of parallel strategy on incremental data from \mathcal{D}_a . We divide dataset \mathcal{D} into 2 smaller splits, \mathcal{D}_1 and \mathcal{D}_2 , calculate the MSTs for \mathcal{D}_a and the complete bipartite graphs between \mathcal{D}_a and each split, then merge these intermediate MSTs along with the initial solution \mathcal{T}_D to find the final MST for $\mathcal{D} \cup \mathcal{D}_a$.

adding new data points. In the second phase, GRIN examine each new data point to determine whether it belongs to leaf nodes of the tentative dendrogram. If it falls into exactly one leaf cluster, then it is labeled as that leaf cluster. Otherwise, the gravity theory is applied to determine which leaf cluster that the point belongs to. Some other techniques on incremental hierarchical clustering include [15, 27, 30].

INCREMENTAL, DISTRIBUTED SINGLE-LINKAGE HIERARCHICAL CLUSTERING USING MAPREDUCE

In this section, we describe our incremental, distributed algorithm for calculating single-linkage hierarchical clustering (SHC) dendrogram.

Hierarchical Clustering

First, we remind the reader about what the hierarchical clustering is. As an often used data mining technique, hierarchical clustering generally falls into two types: agglomerative and divisive. In the first type, each data point starts in its own singleton cluster, two closest clusters are merged at each iteration until all the data points belong to the same cluster. The divisive approach, however, works the process from top to bottom by performing splits recursively. As a typical example of agglomerative approach, Single-linkage hierarchical clustering merges the two clusters with the shortest distance, i.e. the link between the closest data pair (one in each cluster) at each step. Despite the fact that SHC can produce “chaining” effect where a sequence of close observations in different groups cause early merges of these groups, it is still a widely-used analysis tool to conduct early-stage knowledge discovery for its simplicity and quadratic time complexity.

Problem Decomposition

Our goal is to design a scalable and incremental algorithm such that it can not only scale to the large dataset but also accommodate the newly added data incrementally without carrying out clustering from scratch.

Based on the theoretical finding [17] that calculating the SHC dendrogram of a dataset is equivalent to finding the Minimum Spanning Tree (MST) of a complete weighted graph, where the vertices are the data points and the edge weight are the distance between any two points, the incremental SHC clustering problem with a base dataset \mathcal{D} and newly added dataset \mathcal{D}_a can be formulated as follows:

Given a complete weighted graph $\mathcal{G}(\mathcal{D} \cup \mathcal{D}_a)$ induced by the distances between points in $\mathcal{D} \cup \mathcal{D}_a$, design a parallel algorithm to find the MST in the complete weighted graph $\mathcal{G}(\mathcal{D})$ and incrementally solve the MST in $\mathcal{G}(\mathcal{D} \cup \mathcal{D}_a)$.

In this section, we describe how the clustering problem can be decomposed in these two settings. Intuitively, we propose to employ the multi-level paradigm via divide-and-conquer parallelization strategy. The multilevel paradigm is known for its effectiveness when solving very large-scale scientific problems including the top-down divisive clustering (e.g., PDDP [16]) or spectral graph partitioning techniques (e.g., [18]). The main idea of our algorithm is to divide the original problem into a set of non-overlapped subproblems, solve each subproblem and then merge the sub-solutions into an overall solution.

To show the process of problem decomposition, a toy example is illustrated in Figure 1. Given an original dataset \mathcal{D} , we first divided it into two disjoint subsets: \mathcal{D}_1 and \mathcal{D}_2 , thus the complete graph $\mathcal{G}(\mathcal{D})$ is decomposed into to three sub-graphs: $\mathcal{G}(\mathcal{D}_1)$, $\mathcal{G}(\mathcal{D}_2)$ and $\mathcal{G}_B(\mathcal{D}_1, \mathcal{D}_2)$, where $\mathcal{G}_B(\mathcal{D}_1, \mathcal{D}_2)$ is the complete bipartite graph on datasets \mathcal{D}_1 and \mathcal{D}_2 . In this way, any possible edge is assigned to some subgraph, and taking the union of these subgraphs would return us the original graph. This approach can be easily extended to s splits, and leads to multiple subproblems of two different types: s complete subgraph on each split and C_s^2 complete bipartite subgraphs on each pair of splits. Once we complete the dividing procedure and form a set of subproblems, we distribute

these subproblems among multiple processes and apply a local MST algorithm on each of them, the calculated sub-MSTs are then combined to obtain the final solution for the original problem.

One of the main challenges in the design of modern clustering algorithms is that, in many applications, new datasets are continuously added into an already large dataset. As a result, it is impractical to reapply the clustering algorithm on the updated datasets. One way to tackle this challenge is to design a new parallel clustering algorithm that accommodate the update incrementally. Let's denote the newly arrived batch of as \mathcal{D}_a and the original dataset as \mathcal{D} . By adding \mathcal{D}_a into \mathcal{D} , new edges are formed for the complete bipartite on pair $(\mathcal{D}_a, \mathcal{D})$, and the complete subgraph within \mathcal{D}_a , therefore the complete graph on the updated dataset $\mathcal{G}(\mathcal{D} \cup \mathcal{D}_a)$ composes of three subgraphs: $\mathcal{G}_B(\mathcal{D}, \mathcal{D}_a)$, $\mathcal{G}(\mathcal{D}_a)$ and $\mathcal{G}(\mathcal{D})$. Since we already have the MST result on $\mathcal{G}(\mathcal{D})$, only MSTs on $\mathcal{G}_B(\mathcal{D}, \mathcal{D}_a)$ and $\mathcal{G}(\mathcal{D}_a)$ need to be calculated. However, the newly arrived data is usually much smaller than the original dataset, it is aciently to only use one process to calculate the MST on $\mathcal{G}_B(\mathcal{D}, \mathcal{D}_a)$. As shown in Figure 2, one approach is to divide the the original dataset into multiple splits and calculated the sub-MSTs on the complete bipartite subgraphs on each pair of \mathcal{D}_a and the split and the complete subgraph induced on \mathcal{D}_a .

In Figure 1 and 2, we observe that except for the problem decomposition, both cases can be solved by the same local MST algorithm followed by a merge procedure, which leads to the final solution. Therefore, a generalized framework can be designed to handle large datasets with incremental updates.

IncDiSC Algorithm Design

In this section, we present IncDiSC, an incremental, distributed single-linkage hierarchical clustering algorithm.

Algorithm Design

Following the dividing steps described in step 1-8 of Algorithm 1, we break the original problem into multiple much smaller subproblems, a serial MST algorithm can be applied locally on each of them. For a weighted graph, there are three frequently used MST algorithms, namely Borůvka's, Kruskal's and Prim's [7, 20, 25]. Borůvka's algorithm was published back in 1920s. At each iteration, it identifies the cheapest edge incident to each vertex, and then forms the contracted graph which reduces the number of vertices by at least half. Thus, the algorithm takes $O(E \log V)$ time, where E is the number of edges and V is the number of vertices. Kruskal's algorithm initially creates a forest with each vertex as a separate tree, and iteratively selects the cheapest edge that doesn't create a cycle from the unused edge set to merge two trees at a time until all vertices belongs to a single tree. Both of these two algorithms require all the edge weights available in order to select the cheapest edge either for every vertex in the entire graph at each iteration. By contrast, Prim's algorithm starts with an arbitrary vertex as a MST root and then grows one vertex at a time until it spans all the vertices in the graph. At each iteration, it only needs one vertex's local information to proceed. Moreover, given a complete weighted graph, Prim's algorithm only takes $O(V^2)$ time and $O(V)$

Algorithm 1 IncDiSC, an incremental, distributed SHC algorithm

INPUT: a base dataset \mathcal{D} , a newly added dataset \mathcal{D}_a , a minimum spanning Tree \mathcal{T} induced on \mathcal{D} , a merging parameter K
OUTPUT: a MST \mathcal{T}' for $\mathcal{D} \cup \mathcal{D}_a$

```

1: if  $\mathcal{T} == \emptyset$  then
2:   Divide  $\mathcal{D}$  into  $s$  roughly equal-sized splits,
      $\mathcal{D}_1, \mathcal{D}_1, \dots, \mathcal{D}_s$ 
3:   Form  $\mathcal{C}_s^2$  complete bipartite subgraphs and  $s$  complete
     subgraphs
4: else
5:   Divide  $\mathcal{D}$  into  $t$  roughly equal-sized splits,
      $\mathcal{D}_1, \mathcal{D}_1, \dots, \mathcal{D}_t$ 
6:   Form  $t$  complete bipartite subgraphs and a complete
     subgraph
7: end if
8: Use Prim's algorithm to compute the sub-MST on
     each subgraph
9: repeat
10:  Taking the sub-MSTs and  $\mathcal{T}$ , merge every  $K$  of
     them using the idea of Kruskal's algorithm
11: until one MST remains
12: return the final MST  $\mathcal{T}'$ 

```

space complexity, lending itself a good choice for the local MST algorithm.

As mentioned earlier, we have two types of subproblems: complete weighted graph and complete bipartite graph. For the first type of subproblem, we start with the first vertex v_0 in the vertex list just for convenience. While we populate all its edge weights by calculating distance from v_0 to every other vertex, we track the cheapest edge and emit the corresponding edge to the reducer in MapReduce framework (in this way, we don't need to store the MST explicitly). v_0 is then removed from the vertex list and the other endpoint of the emitted edge is selected to be the next starting vertex. This process is repeated until all the vertices are added to the tree. Thus, our algorithm maintains quadratic time complexity and linear space complexity.

The other type of subproblem is the complete bipartite subgraph between two disjoint data splits, denoted as the left and right split. Different from the complete subgraph case, we need to maintain an edge weight array for each split respectively. To start, we select the first vertex v_0 in the left split, populates an edge weight array from v_0 to every vertex in the right split, record the cheapest edge (v_0, v_t) . In the next iteration, we populate another edge weight array from v_t to every vertex in the left split except for v_0 . Then the cheapest edge is selected from both edge weight arrays. The endpoint of the cheapest edge (which is neither v_0 nor v_t) is selected as the next starting vertex, and the same process can be iterated until the tree spans all the vertices. The procedure takes $O(mn)$ time complexity and $O(m+n)$ space complexity, where m, n are the sizes of the two disjoint sets.

Step 10 in Algorithm 1 then iteratively merges all the intermediate sub-MSTs and the precalculated \mathcal{T} to obtain the over-

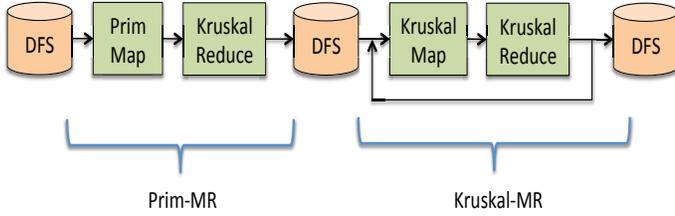


Figure 3: IncDiSC framework using MapReduce.

all solution. In extreme case, all the sub-MSTs and \mathcal{T} can be combined all at once using one process, however, this incurs huge communication contention and computational load; rather, we extend the merge procedure into multiple iterations by introducing configurable parameter K such that every K intermediate MSTs are merged at each iteration and it terminates when only one MST remains.

In order to efficiently combine these partial MSTs, we use union-find (disjoint set) data structure to keep track of the component to which each vertex belongs [12]. Recall the way we form subgraphs, most neighboring subgraphs share half of the data points. Every K consecutive subgraphs more likely have a fairly large portion of overlapping vertices. Thus, by combining every K sub-MSTs, we can detect and eliminate incorrect edges at an early stage, and reduce the overall communication cost for the algorithm. The communication cost can be further optimized by choosing the right K value with respect to the size of dataset, which we will discuss in the next section.

IncDiSC bcpReduce

MapReduce has emerged as one of the most frequently used parallel programming model for processing large-scale datasets since it was first proposed in 2004 by Dean and Ghemawat [14]. Its open source implementation, Hadoop, has become the de facto standard for both industry and academia. In the following, we describe in details the implementation of our proposed algorithm using Hadoop’s MapReduce Framework.

A MapReduce job comprises three consecutive phases: map, shuffle and reduce. The input, output as well as intermediate data, is formatted in $(key, value)$ pairs. In the map phase the input is processed one tuple at a time. All $(key, value)$ pairs emitted by the map phase which have the same key are then aggregated by the MapReduce system during the shuffle phase and sent to the reducer. At the reduce phase, each key, along with all the values associated with it, are processed together. In order for all the values with the same key end up on the same reducer, a partitioning or hash function need to be provided for the shuffle phase. The system then makes sure that all of the $(key, value)$ pairs with the same key are collected on the same reducer.

Figure 3 provides an overall MapReduce diagram for Algorithm 1. The algorithm can be implemented by two types of MapReduce jobs: Prim-MR and Kruskal-MR. The Prim-MR job is executed only once while the Kruskal-MR job can

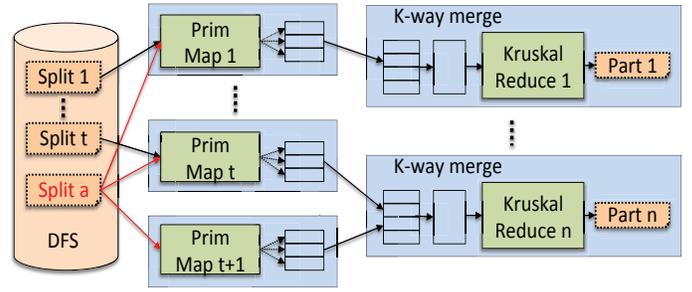


Figure 4: Prim-MR Diagram: Prim-MR is the first MapReduce job in IncDiSC algorithm. For incremental data insertion case, it consists of $t + 1$ Prim maps and n Kruskal reducers, where $n = \lceil (t + 1)/K \rceil$.

be repeated for multiple rounds until the final MST is completed. The aggregated results after a MR job is stored on the distributed file system, e.g. Hadoop distributed file system, as the input for the next available MR job. Prim-MR consists of PrimMap and KruskalReduce. Its detailed implementation is illustrated in Figure 4. The setup in the diagram also can handle the incremental case where the new arrived data split D_a is inserted. The original dataset is divided into t smaller splits. Each split is stored in the built-in SequenceFile format provided by Hadoop. SequenceFile is a flat binary record file with each record being a key-value pair. It is extensively used as Hadoop’s input/output formats. In our case, the input data splits are keyed on the data point’s id and valued on the corresponding feature vector.

In order for a mapper to know which two splits to be read, we initially produce $t + 1$ input files, each of which contains a single integer value gid between 0 and t to represent the subgraph id. Without loss of generality, the subgraph $(t + 1)$ is a complete subgraph while the others are complete bipartite subgraphs. Given a certain graph type, we apply the corresponding Prim’s algorithm accordingly. For the case where there is no precalculated MST available for the original dataset, the decomposition in Figure 1 is adopted in the Prim-MR framework.

Each complete bipartite subgraph is assigned to a mapper. However, the complete subgraph only has half as many vertices as the complete bipartite subgraph. In order to keep the load balance, we assign two complete subgraphs to one mapper. On each mapper, a local sequential MST algorithm is implemented to calculate the MST on subgraph gid . As described in Algorithm 2, PrimMap’s input is keyed on gid and valued on data point. The algorithm first populate two arrays S_1 and S_2 . S_2 may be empty if the number of complete subgraphs is odd. Once both arrays get populated, we start to calculate the MST. As described previously, given a complete graph, the local MST algorithm starts with a single-node tree, and then augments the tree one vertex at a time by greedily selecting the cheapest edge among all the edges we have calculated so far. Instead of releasing all the edges after the MST is complete, the edges can be emitted one by one as they are generated, reducing I/O and network contention.

Algorithm 2 PrimMap: calculate the MST for a complete bipartite subgraph or a complete subgraph

```

1: map(Int gid, Point value)
2:   Point[] S1, S2;
3:   cnts = 0;
4:   if (cnts < |S1|) do
5:     S1.append(value)
6:     cnts++; return;
7:   else if (cnts < (|S1| + |S2|)) do
8:     S2.append(value)
9:     cnts++; return;
10:  end
11:  if (isBiPartite) do
12:    PrimBipartite mst = new PrimBipartite(S1, S2)
13:    mst.emitMST()
14:  else do
15:    PrimComplete mst = new PrimComplete(S1)
16:    mst.emitMST()
17:    if (S2 is not NULL) do
18:      mst = new PrimComplete(S2)
19:      mst.emitMST()
20:    end
21:  end

```

As PrimMap emits the edge one at a time, the output is spilled into multiple temporary files on the local machine in a sorted order, and transferred to the designated reducer based on the partitioner. Before passing to the reducer, the files are concatenated in the sorted order and merged into a single input file. This is called data shuffle or sort-and-merge stage. Hadoop sorts keys by default and provides a secondary sorting mechanism, with which we can design a composite key containing graph id and the edge weight. In this way, K -way can be implicitly implemented in Hadoop under the hood without actual realization in the KruskalReduce.

In Kruskal-MR job, the map function is essentially an identity map which just passes through (key, value) pairs as they are. The reduce function reuses KruskalReduce to combine any intermediate MSTs and precalculate \mathcal{T} . The Kruskal-MR job is iterated until one MST remains.

EXPERIMENTAL RESULTS

Here, we discuss the experimental results on the implementation of IncDiSC algorithm using both synthetic and real-world datasets. We evaluate the scalability of the algorithm on the dataset with various size and dimensions as well as the data shuffle and I/O patterns in each MapReduce round.

We conduct the experiments on Jesup [2], a Hadoop cluster at NERSC. Jesup has 80 compute nodes each of which is quad-core Intel Xeon X5550 “Nehalem” 2.67 GHz processors (eight cores per node) with 24 GB of memory per node. All the nodes are interconnected by 4X QDR InfiniBand technology that provides 32 Gb/s of point-to-point bandwidth for data communication and I/O. However, computer nodes in Jesup have no local disks, which implies no data locality can be leveraged from Hadoop distributed file system. Despite this downside caused by the system specifics, our experiments

Table 1: Structural properties of the synthetic-cluster, synthetic-random, and millennium-run-simulation testbed. The data size is measured in SequenceFile format which is compressed by GZIP.

Name	Points	Dimensions	Size (MByte)
<i>clust20k</i>	20k	5, 10	1.3, 2.1
<i>clust100k</i>	100k	5, 10	6.6, 10
<i>clust500k</i>	500k	5, 10	32, 49
<i>rand20k</i>	20k	5, 10	1.3, 2.1
<i>rand100k</i>	100k	5, 10	6.6, 10
<i>rand500k</i>	500k	5, 10	32, 49
<i>DGalaxiesBower2006a (db)</i>	1m	3	51
<i>MPAHaloTreesMhalo (mm)</i>	1m	3	51
<i>MPAGalaxiesBertone2007 (mb)</i>	1m	3	51
<i>MPAGalaxiesDeLucia2006a (md)</i>	1m	3	51

still show the real computation and communication behaviors as in many other clusters.

Datasets

We evaluate IncDiSC using sixteen datasets, which are divided into three categories: *synthetic-cluster*, *synthetic-random* and *millennium-run-simulation*. The first two categories are synthesized by using the IBM synthetic data generator [4,24]. With three different numbers of data points and two different dimensions, we generate six *synthetic-cluster* datasets, in which a random number of centroids are selected first and data points are added randomly to these centroids. And in six *synthetic-random* datasets, points in each dataset are uniformly distributed.

The third category *millennium-run-simulation* consists of four real-world datasets, MPAGalaxiesBertone2007 (*mb*) [6], MPAGalaxiesDeLucia2006a (*md*) [13], DGalaxiesBower2006a (*db*) [8], and MPAHaloTreesMhalo (*mm*) [6] which are taken from the Galaxy and Halo databases (as the name specified). Because we only take the first 3 dimensions (particle coordinates) from each dataset, we have randomly selected **1 million** points from these datasets. Our testbed contains up to **1 million** data points and each data point is a vector of up to **10** dimensions. Table 1 shows structural properties of the dataset.

Performance

Scalability

We first evaluate how well our algorithm scales from 10 to 300 computer cores on twelve synthetic datasets with different K values, where K is the number of subgraphs that can be merged at one reducer.

The speedup on p cores is defined as $S = p_0 \frac{t_{p_0}}{t_p}$, with p_0 being the minimum computer cores that we conduct our experiments with, and t_p being the DiSC’s execution time on p cores. Figure 5 (a) presents the speedup result for *clust20k* with 10 dimensions. “Total” measures the algorithm’s entire execution time. It comprises “Prim” measuring the runtime for Prim-MR job, and “Kruskal” measuring the runtime for a series of Kruskal-MR jobs until the algorithm completes.

The performance has no improvement or even turns worse as we scale out for *clust20k_10*. This makes sense since small

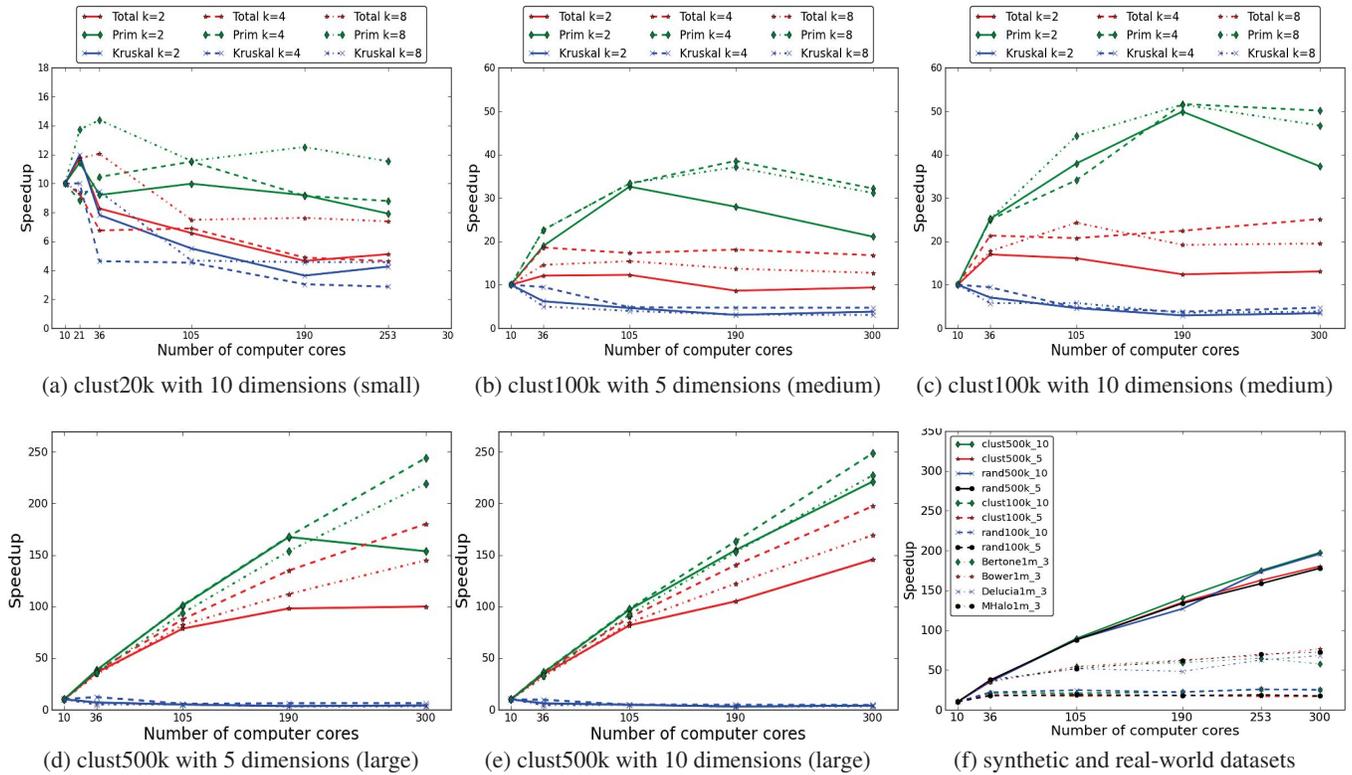


Figure 5: Speedup on the synthetic datasets using 10-300 computer cores.

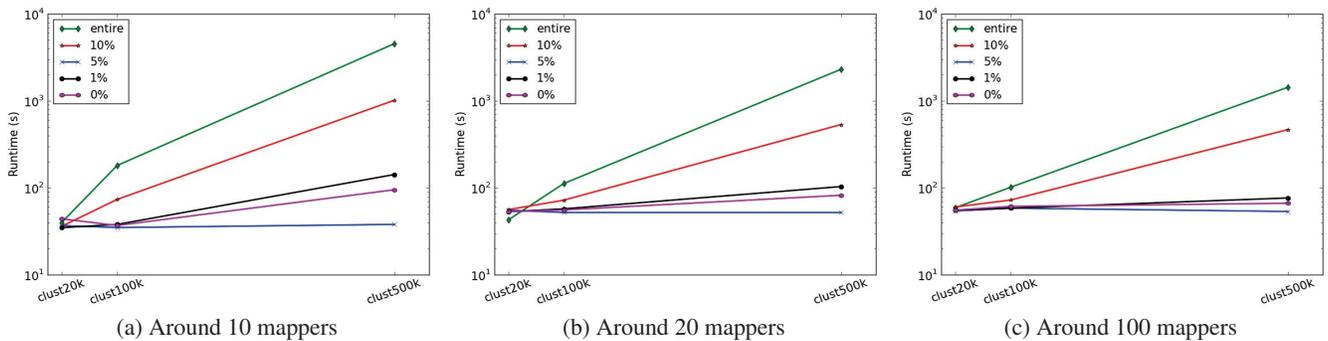


Figure 6: Runtime of DiSC vs. runtime of IncDiSC.

datasets can fit in one or a few machines' memory, the overhead introduced by unnecessary multiple MapReduce iterations would offset the computational gain from data parallelism.

For both medium and large datasets, Figure 5 (b) – (e) demonstrates a nice linear speedup until the number of compute cores increases beyond a certain number. For the medium-sized dataset, Figure 5 (c) shows that even with high dimensionality, the speedup starts to drop regardless of the number of dimensions, when the number of processes is beyond 190, which corresponds to 20 splits on the original data, each

with $5k$ data points. However, with $K = 2$, we approximately need 7 MR iterations for the entire algorithm, thus the communication cost is no longer negligible. As illustrated in Figure 5 (e) the linear speedup sustains up to 300 compute cores for large datasets with high dimensionality. Among all four plots, $K = 4$ consistently outperforms $K = 2$ or 8. This is because K not only affects the number of MapReduce rounds, but also the number of reducers at the merge phase at each round. Increasing K value leads to a smaller number of iterations and smaller degree of parallelism. The value of

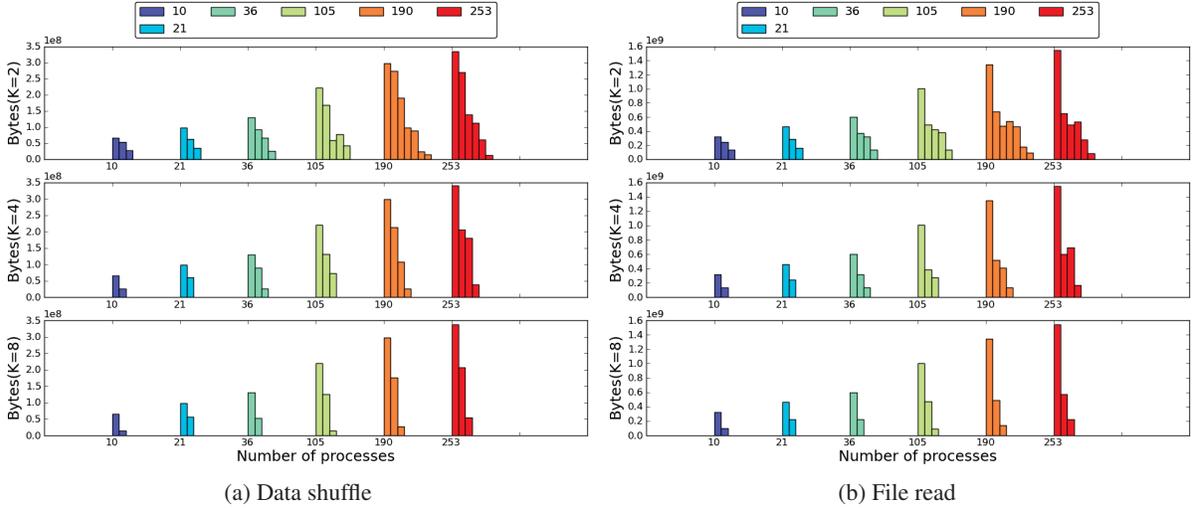


Figure 7: Data Pattern for IncDiSC algorithm: the amount of data bytes in the stages of data shuffle and file read. (File write is omitted here and it exhibits the same trend as file read.)

4 seems to be a sweet spot achieving a reasonable trade-off between these two effects.

Figure 5 (f) summarizes the speedup results on both synthetic and real-world datasets. As expected, the number of objects in the dataset significantly influences the speedups (bigger datasets show better scalability), and the dimensionality is another factor that affects the performance. The type of dataset hardly makes any difference in our algorithm as we use Euclidean distance as the edge weight measure, and the distribution of data points has no impact on the computational complexity in calculating distances.

Incremental Insertion

We evaluate the efficiency of IncDiSC using *synthetic-cluster*'s three datasets with 10 dimensions. On each synthetic dataset, we perform 0.5%, 0.1%, 5%, and 10% random insertions and compare the runtime of IncDiSC algorithm on the entire updated dataset from the scratch with the average runtime of IncDiSC solely on the random updates. The results with different granularity of problem decomposition are depicted in Figure 6. We first divide the original dataset into 10 splits, each pairing with the insertion split to form 10 complete bipartite subgraphs in addition to a complete subgraph on the insertion alone. To be consistent with the case in which we reapply IncDiSC on the entire updated dataset, we accordingly partition the accumulated dataset into 5 splits which leads to 10 complete bipartite subgraphs and 5 complete subgraphs. Given this setup, we conduct experiments with 10 mappers as shown in Figure 6, and observe the speedup factors of about 1, 5, 120 on datasets of $20k$, $100k$, $500k$ data points respectively for small insertions up to 5%, and the speedup degrades to 5 when the insertion is 10% of the original data. Such a large percentage of addition usually would not appear in the real-world because the base dataset is already very large while the incremental update is batched and processed over small time interval. A similar speedup ob-

servation can be made when evaluating performance on finer decomposition on 20 and 100 mappers with about 45X and 27X speedup for 5% insertion respectively. The speedup factor doesn't increase as the number of mappers increases. A possible reason is that each complete bipartite subgraph is already fairly small when we split the original dataset into 5 partitions. Making finer decomposition would not buy us more data parallelism, it leads to more Kruskal-MR iterations which comes with MapReduce framework overhead, such as job setup, tear-down, data shuffle, etc. For large-scale and high-dimensional datasets, IncDiSC saves the runtime by two orders of magnitude over the naive implementation that works on the entire updated dataset without incorporating the priori clustering results.

I/O and Data Shuffle

In this section, we evaluate the data patterns with respect to MapReduce metrics, including file read, file write and data shuffle from mapper to reducer per iteration. Recall when we form the complete bipartite subgraphs, each split need be paired with every other split. Therefore, the amount of data read from the disk is linear to the number of splits, which is approximate the square root of the number of mappers. In Figure 7, each bar represents a MapReduce round, and bars in the same color represent a series of MR rounds that IncDiSC algorithm requires to find the MST given a certain number of computer cores. For example, the first bar represents Prim-MR job in IncDiSC algorithm. Figure 7 (a) illustrates the increasing trend of the amount of data shuffle from mapper to reducer. Notably, as we scale up the number of processes, the number of MapReduce rounds increases. However, the data is dramatically reduced after the first Prim-MR job by almost 2 orders of magnitude, which verifies our claim that incorrect edges are pruned at a very early stage. The same trend is observed for file read at mapper's input and file write at reducer's output. After the first iteration, the amount of

data shuffle and I/O is proportional to the number of vertices residing in the merged subgraphs, and the amount of vertices decreases by approximately K times due to the deduplication effect at the KruskalReduce's merging process. Figure 7 reveals that the number of iterations decreases with large K , so does the amount of the data. This finding also corresponds with speedup chart that 4-way merge outperforms 2- and 8-way merges because it provides a good trade-off between the number of MR rounds and the degree of parallelism per round.

CONCLUSION

In this paper, we present IncDiSC, an incremental, distributed algorithm for single-linkage hierarchical clustering algorithm to overcome the data dependency and algorithm complexity challenges in a unified framework. IncDiSC can not only scale to the large dataset, but also incorporate the incremental accumulation of the new data. We evaluated IncDiSC empirically using both synthetic and real-world datasets, and observed that it achieves a speedup up to 200 on 300 computer cores and two orders of magnitude speedup for up to 5% the insertion update. Experimental results have shown that IncDiSC reduces the amount of data shuttle dramatically at early iterations. Future work on IncDiSC may involve efforts to tackle the deletion update such that the algorithm can be adapted to much wide modern applications.

Acknowledgment

This work is supported in part by the following grants: NSF awards CCF-1029166, ACI-1144061, IIS-1343639, and CCF-1409601; DOE award DESC0007456; AFOSR award FA9550-12-1-0458; NIST award 70NANB14H012.

REFERENCES

1. http://www.facebook.com/blog/blog.php?topic_id=185341929641. [Online].
2. <http://www.nersc.gov/users/computational-systems/testbeds/jesup>. [Online].
3. SLINK: An optimally efficient algorithm for the single-link cluster method. *The Computer Journal*, 1 (1973).
4. Agrawal, R., and Srikant, R. Quest Synthetic Data Generator. *IBM Almaden Research Center* (1994).
5. Alizadeh, A., Eisen, M., Davis, R., Ma, C., Lossos, I., Rosenwald, A., Boldrick, J., Sabet, H., Tran, T., Yu, X., Ji, P., Yang, L., Ge, M., Moore, T., Hudson, J., Lu, L., Tibshirani, R., Sherlock, G., Chan, W., Greiner, T., Weisenburger, D., Armitage, J., Warnke, R., Levy, R., Wilson, W., Grever, M., Byrd, J., Botstein, D., Brown, P., and Staudt, L. Distinct types of diffuse large B-cell lymphoma identified by gene expression profiling. *Nature* (2000).
6. Bertone, S., De Lucia, G., and Thomas, P. The recycling of gas and metals in galaxy formation: predictions of a dynamical feedback model. *Monthly Notices of the Royal Astronomical Society* 379, 3 (2007), 1143–1154.
7. Boruvka, O. O Jistém Problému Minimálním (About a Certain Minimal Problem) (in Czech, German summary). *Práce Mor. Průrodoved. Spol. v Brne III* 3 (1926).
8. Bower, R., Benson, A., Malbon, R., Helly, J., Frenk, C., Baugh, C., Cole, S., and Lacey, C. Breaking the hierarchy of galaxy formation. *Monthly Notices of the Royal Astronomical Society* 370, 2 (2006), 645–655.
9. Cathey, R. J., Jensen, E. C., Beitzel, S. M., Frieder, O., and Grossman, D. Exploiting parallelism to support scalable hierarchical clustering. *Journal of the American Society for Information Science and Technology* 58, 8 (2007), 1207–1221.
10. Chang, D.-J., Kantardzic, M. M., and Ouyang, M. Hierarchical clustering with CUDA/GPU. In *ISCA PDCCS*, J. H. Graham and A. Skjellum, Eds., ISCA (2009), 7–12.
11. Chen, C., Hwang, S.-C., and Oyang, Y.-J. An incremental hierarchical data clustering algorithm based on gravity theory. In *Proceedings of the 6th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining*, PAKDD (London, UK, 2002), 237–250.
12. Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.
13. De Lucia, G., and Blaizot, J. The hierarchical formation of the brightest cluster galaxies. *Monthly Notices of the Royal Astronomical Society* 375, 1 (2007), 2–14.
14. Dean, J., and Ghemawat, S. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, USENIX Association (Berkeley, CA, USA, 2004), 10–10.
15. Gurrutxaga, I., Arbelaitz, O., Martín, J. I., Mugerza, J., Pérez, J. M., and Perona, I. n. SIHC: A Stable Incremental Hierarchical Clustering Algorithm. In *ICEIS (2)'09* (2009), 300–304.
16. Heisele, B., Serre, T., Mukherjee, S., and Poggio, T. Feature reduction and hierarchy of classifiers for fast object detection in video images. *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on* 2 (2001), 18.
17. Hendrix, W., Patwary, M. M. A., Agrawal, A., keng Liao, W., and Choudhary, A. Parallel hierarchical clustering on shared memory platforms. In *HiPC* (2012), 1–9.
18. Hofmeyr, S. A., Forrest, S., and Somayaji, A. Intrusion detection using sequences of system calls. *Journal of Computer Security* 6 (1998), 151–180.
19. Jain, A. K., Murty, M. N., and Flynn, P. J. Data clustering: A review, 1999.

20. Kruskal, J. B. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society* 7, 1 (Feb. 1956), 48–50.
21. Lattanzi, S., Moseley, B., Suri, S., and Vassilvitskii, S. Filtering: a method for solving graph problems in mapreduce. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA (2011), 85–94.
22. Liu, L., Rui, Y., Sun, L., Yang, B., Zhang, J., and Yang, S.-Q. Topic mining on web-shared videos. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)* (2008), 2145–2148.
23. Madeira, S. C., and Oliveira, A. L. Biclustering algorithms for biological data analysis: A survey. *IEEE/ACM Trans. Comput. Biol. Bioinformatics* 1, 1 (Jan. 2004), 24–45.
24. Pisharath, J., Liu, Y., keng Liao, W., Memik, G., Choudhary, A., and Dubey, P. NU-MineBench 3.0.
25. Prim, R. C. Shortest connection networks and some generalizations. *Bell System Technology Journal* (1957).
26. Rastogi, V., Machanavajjhala, A., Chitnis, L., and Sarma, A. D. Finding connected components on map-reduce in logarithmic rounds. *CoRR abs/1203.5387* (2012).
27. Sahoo, N., Callan, J., Krishnan, R., Duncan, G., and Padman, R. Incremental hierarchical clustering of text documents. In *Proceedings of the 15th ACM International Conference on Information and Knowledge Management, CIKM '06* (New York, NY, USA, 2006), 357–366.
28. Surdeanu, M., Turmo, J., and Ageno, A. A hybrid unsupervised approach for document clustering. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining* (2005), 685–690.
29. Wang, S., and Dutta, H. Parable: A parallel random-partition based hierarchical clustering algorithm for the mapreduce framework. *Technical Report, CCLS-11-04* (2011).
30. Widyanoro, D. H., Ioerger, T. R., and Yen, J. An incremental approach to building a cluster hierarchy. In *IEEE International Conference on Data Mining* (2002), 705–708.

Throughput Studies on an InfiniBand Interconnect via All-to-All Communications

Nil Mistry
Department of Mathematics
and Statistics, University of
Connecticut
nil.mistry@uconn.edu

Jackie Yanchuck
Department of Mathematics,
Seton Hill University
yan6374@setonhill.edu

Jordan Ramsey
Department of Computer
Science and Electrical
Engineering, UMBC
jramsey3@umbc.edu

Xuan Huang
Department of Mathematics
and Statistics, UMBC
hu6@umbc.edu

Benjamin Wiley
Department of Mathematics
and Statistics, University of
New Mexico
bwiley01@unm.edu

Matthias K. Gobbert
Department of Mathematics
and Statistics, UMBC
gobbert@umbc.edu

ABSTRACT

Distributed-memory clusters are the most important type of parallel computer today, and they dominate the TOP500 list. The InfiniBand interconnect is the most popular network for distributed-memory compute clusters. Contention of communications across a switched network that connects multiple compute nodes in a distributed-memory cluster may seriously degrade performance of parallel code. This contention is maximized when communicating large blocks of data among all parallel processes simultaneously. This communication pattern arises in many important algorithms such as parallel sorting. The cluster tara in the UMBC High Performance Computing Facility (HPCF) with a quad-data rate InfiniBand interconnect provides an opportunity to test if the capacity of a switched network can become a limiting factor in algorithmic performance. We find that we can design a test case of a problem involving increasing usage of memory that does not scale any more on the InfiniBand interconnect, thus becoming a limiting factor for parallel scalability. However, for the case of stable memory usage of the problem, the InfiniBand communications get faster and will not inhibit parallel scalability. The tests in this paper are designed to involve only basic MPI commands for wide reproducibility, and the paper provides the detailed motivation of the design of the memory usage needed for the tests.

Author Keywords

InfiniBand interconnect, All-to-All communications, network contention, scalability studies, MPI

ACM Classification Keywords

I.6.1 SIMULATION AND MODELING (e.g. Model Development). : Performance

INTRODUCTION

The TOP500 list at www.top500.org of the world's most powerful supercomputers has been dominated by distributed-memory clusters for many years by now.

Distributed-memory clusters require a network for communication among all nodes. The high-performance InfiniBand interconnect is currently the most popular network to communicate among the parallel processes on several nodes in distributed-memory compute clusters.

Information transferred among nodes may stress communication across the InfiniBand network, both in relation to the size of the data being sent and the number of nodes being considered. As communication increases, contention among the parallel processes will stress the network due to the massive transfer of data among compute nodes. At extremely high levels of contention, the network is expected to eventually fail to process inter-node communication efficiently. This work studies this effect by creating the maximum possible contention by simultaneous communication among all processes, created by All-to-All communication commands. The tests in this paper are designed to involve only basic MPI commands for wide reproducibility, and the paper provides the detailed motivation of the design of the memory usage needed.

To accomplish sufficient inter-node stress within the network in an algorithmically realistic way, we implemented a parallel sorting function which transfers the local portion of a user-defined number of n pieces of data among the p processes. Before the communications, each node holds a portion of an array of data that is sorted locally, but may contain portions that should reside on other processes. Using All-to-All commands, each individual node sends the appropriate portions of data to all other nodes in the network simultaneously, thus maximizing inter-node communication stress. Our results show that, for constant *global* memory, as the number of processes increase, the runtime in fact improves as the network contention decreases under All-to-All communication. Alternately, as global memory usage is increased maximally, that is, for constant *local* memory, as the number of processes increase, the runtime deteriorates as the network contention increases under All-to-All communication. This test demonstrates that stress on the InfiniBand network can be created that will eventually

limit the scalability of parallel algorithms that use All-to-All communications as building blocks. The situation of constant local memory occurs in weak scalability studies, while the situation of constant global memory occurs in strong scalability studies. Therefore, both are relevant in practice, and the conclusion whether the InfiniBand interconnect can become a limiting factor to parallel scalability will depend on the algorithmic structure of the code.

The sorting example used as prototypical basis for our experiments is inspired by [8, Chapter 10] that uses it to introduce the `MPI_Alltoall` and `MPI_Alltoallv` commands. Other works that test the usage of these commands over InfiniBand are [1, 3–6, 11]. Experiments designed to stress the network data transfer over InfiniBand are described in [2, 13], and [10] reports using FFT and RandomAccess to stress the bandwidth and latency of the Infiniband.

BACKGROUND

Computational Environment and InfiniBand Interconnect

The studies were performed on the cluster tara in the UMBC High Performance Computing Facility (HPCF). All details of the cluster tara and in particular about its InfiniBand interconnect are posted on the webpage www.umbc.edu/hpcf. Various performance studies using tara are available as technical reports, for instance [9] that compares performance by two implementations of MPI. Following [9], we use the MPI implementation MVAPICH2.

The cluster tara has 86 nodes, comprising 82 compute nodes, 2 develop nodes, 1 user node, and 1 management node. Each node has two quad-core Intel Nehalem X5550 processors (2.66 GHz, 8192kB cache) and 24 GB of local memory. All components of tara are connected by a quad-data rate InfiniBand interconnect.

InfiniBand is a connection that allows for high-speed data transfers from computers to input/output devices [12]. It is a switched fabric communication link, meaning that it connects the nodes to each other via switches. In computer networks, switches receive data sent from one device and direct the data to only the device(s) which is (are) meant to receive the data [12]. This allows for more secure and potentially faster data transfers between multiple devices. Using the InfiniBand communication network, there is very low latency (1.2 μ s to transfer a message between two nodes), and wide bandwidth up to 3.5 GB (28 Gb) per second.

It is intuitive to hypothesize that as the number of processes on which a parallel job is run increases, the communication between processes will become slower and may bottleneck because more processes need to communicate with each other than when the number of processes is small. However, many times, commercial manufacturers attempt to avoid this occurrence by using methods such as virtual channels and adaptive routing. Adaptive routing, as apposed to merely routing, allows

nodes to reroute the path that data is sent based on network fluctuations, such as congestion at one node. When a problem is encountered while transferring data, information is sent to the appropriate nodes, and new paths to send data that avoid problem areas are created [12]. Virtual channels were created in order to alleviate the issue of deadlock, and also decrease network latency and throughput. Though these methods are commonly used in parallel computing technology to solve many communication issues that arise, their effects on performance have not been rigorously studied. Therefore, it is difficult to determine when inter-job communication will become a performance issue. Our experiments study this issue. In order to study the effects of inter-job communication on job performance, our team implemented a sorting algorithm which requires communication between all parallel processes.

Leaf Modules

Each leaf module in the InfiniBand switch in the cluster has 18 ports. Two leaf modules currently have complete sets of 18 compute nodes attached to them. The remaining leaf modules contain other nodes that are not part of the partition of compute nodes (such as the develop nodes or components of the storage system) or have a defective node among its connections. We can control the choice of leaf module by explicitly requesting nodes for our jobs by name. Therefore, in this study, our team focuses on how contention is effected both within one leaf module and contention over two leaf modules. Considering this network contention provides insight into whether parallel algorithms that send large blocks of data via All-to-All communications result in contention first over two leaf modules or whether there is contention using nodes located within just one leaf module. More importantly, our conclusions answer the question regarding whether implementation of parallel code requiring All-to-All communications of large data seriously degrades performance.

The 18 ports in one leaf module are arranged evenly in two rows of ports; that is, nine ports are located in the first row and nine nodes are located in a second row. Each of the nine ports are separated in three groups of three ports. Our team studied contention by running several tests by requesting specific nodes, starting with three nodes that form one group on the leaf module, then testing nine nodes or one row in the leaf module, and finally extending this process to the whole leaf module with 18 nodes, and then across two leaf modules with 36 nodes. By keeping the node assignments as tightly together as possible in this way, we give the InfiniBand interconnect the best opportunity to overcome contention, since the more local communications can profit from the most complete set of interconnection.

METHODOLOGY

All-to-All Communications

An All-to-All communication simultaneously sends and receives data between all parallel processes in one call.

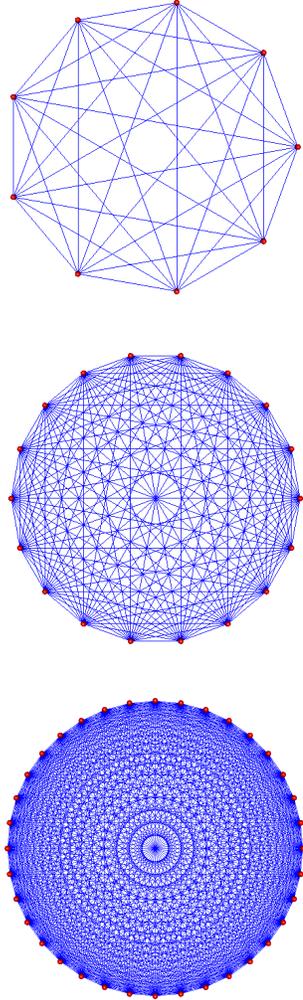


Figure 1. Network schematics for All-to-All communication between $N = 9, 18, 36$ nodes.

Since it is eventually not possible to have physical cable connections between all possible pairs of ports in the InfiniBand switch and its leaf modules, All-to-All commands necessarily lead to contention between all required pairwise communications. The network schematics in Figure 1 give a visual impression of how many cables would be needed to connect $N = 9, 18, 36$ nodes, respectively. An All-to-All communication command sends the j^{th} block of its input array from Process i to Process j and receives it into the i^{th} block of the output array on Process j . MPI has two All-to-All communication commands: `MPI_Alltoall` and `MPI_Alltoallv`. The former command sends the same amount of data between all processes, while the latter one can send variable (hence the letter “v” at the end of the command name) amounts of data between all processes [8]. To test the InfiniBand network, we will maximize the contention by communicating the largest block sizes possible. Thus, in our studies, also the variable version `MPI_Alltoallv` will send (by choosing an appropriate example data set) the

same amount of data between all processes, since that maximizes contention between messages.

Experimental Design

In order to effectively stress inter-job communication, our team implemented a sorting function which transfers data among all nodes within the InfiniBand network utilizing the MPI commands `MPI_Alltoall` and `MPI_Alltoallv`. The idea of the algorithm is inspired by [8, Chapter 10] that uses a similar example to introduce these MPI commands. The data structure is given by n numbers, which are distributed onto the p parallel processes. Only local arrays of length $l_n := n/p$ are stored, never a global array of length n . Only the minimum number of arrays are used in the algorithm, namely one vector *unsorted* that holds the unsorted data originally and one vector *sorted* that holds the sorted data at the end of the algorithm. These two vectors have length l_n on each parallel MPI process. In [8], the algorithm has four steps: (i) The data in *unsorted* is sorted locally on each process (by any serial method of choice); while the numbers in *unsorted* are now sorted, they may contain components that need to be sent to the other process, thus creating the need for All-to-All communications. (ii) An `MPI_Alltoall` call communicates a single integer among all process pairs that informs the process pairs, how many pieces of data need to be sent and received among them in the next step. (iii) An `MPI_Alltoallv` call communicates the appropriate portions of the local *unsorted* vector on each process to the appropriate portions of the local *sorted* vector on each process. (iv) The numbers in the received *sorted* vector then still need local sorting (by any serial method) to obtain the final result of the algorithm, in which the *sorted* vectors — if they were concatenated from all processes — are globally sorted.

To focus entirely on the effect of the communications on the timings, we choose a sample dataset, in which neither of the local sort algorithms in steps (i) or (iv) above are needed. Moreover, since the goal is to stress the network by having as much simultaneous parallel communication as possible, we design the dataset in the initial *unsorted* vector to have an equal number of components that need to be sent to all other processes. That is concretely, out of the $l_n = n/p$ numbers in *unsorted* on one parallel process, the same block length of l_n/p components needs to be sent to each of the p processes.

The idea is best understood by a concrete example of $n = 48$ numbers, given as numbers $1, 2, \dots, 48$, distributed to $p = 4$ processes, with ID numbers $0, 1, 2, 3$ in MPI counting, displayed in the matrix *unsorted* =

$$= \begin{bmatrix} 1, 2, 3, & 13, 14, 15, & 25, 26, 27, & 37, 38, 39 \\ 4, 5, 6, & 16, 17, 18, & 28, 29, 30, & 40, 41, 42 \\ 7, 8, 9, & 19, 20, 21, & 31, 32, 33, & 43, 44, 45 \\ 10, 11, 12, & 22, 23, 24, & 34, 35, 36, & 46, 47, 48 \end{bmatrix}.$$

The $p = 4$ rows in this matrix list the $l_n = n/p = 48/4 = 12$ numbers each that are initially on the Pro-

cesses 0, 1, 2, 3, respectively. We note that the numbers in each row above are already locally sorted, thus not requiring a local sort of step (i) of the algorithm. To achieve a globally sorted vector, stored in local vector *sorted* on each process, requires for this sample dataset the communication of a block length $l_n/p = 12/4 = 3$ of numbers among all pairs of processes. For example for Process 0 (data in first row of the matrix), the group of numbers 13, 14, 15 needs to be sent to Process 1, which — coming from Process 0 — will show up as the first numbers in vector *sorted* on Process 1. This communication gives the result that can be summarized in the matrix *sorted* =

$$= \begin{bmatrix} 1, 2, 3, & 4, 5, 6, & 7, 8, 9, & 10, 11, 12 \\ 13, 14, 15, & 16, 17, 18, & 19, 20, 21, & 22, 23, 24 \\ 25, 26, 27, & 28, 29, 30, & 31, 32, 33, & 34, 35, 36 \\ 37, 38, 39, & 40, 41, 42, & 43, 44, 45, & 46, 47, 48 \end{bmatrix},$$

which lists in each row the numbers in *sorted* on the Processes 0, 1, 2, 3, respectively; notice the group 13, 14, 15 at the start of the second row for Process 1. We observe that the numbers in each row of this matrix are sorted and no local sort of the vectors *sorted* on each process in step (iv) of the algorithm is needed.

Memory Predictions

To stress the network as much as possible, we need to make the amount of data communicated between each pair of parallel processes as large as possible. In the example dataset designed so far, this amount of data is simply a block length l_n/p of numbers, which is thus given indirectly by choosing n and p . We introduce now another independent variable m that allows to control the amount of this data independently from n and p . Namely, in place of each number in the example for the arrays *unsorted* and *sorted* we use a struct that contains an array of m double-precision numbers. We can now think of the numbers in *unsorted* and *sorted* as indices into an array of structs, and communicating each struct requires the sending and receiving of m doubles. In other words, in place of communicating l_n/p numbers between process pairs, we communicate l_n/p many vectors of m double-precision numbers, called a block size l_n/p of m -vectors for short.

The benefit of introducing m is that we can now explicitly control the memory requirement of the arrays by choosing m . The two local vectors, *unsorted* and *sorted*, are the only major variables in memory. Since each node on the cluster tara has 24 GB of memory, total local memory must be less than 24 GB per node. To comfortably stay within this memory also on one node, the vectors are chosen as less than 10 GB each to insure that memory does not become a problem. In Table 1, we specialize our memory calculations to use the maximum possible number of 8 parallel processes on each compute node, which maximizes contention on each node for the All-to-All communications among its local processes and contention when all local processes access the InfiniBand cable at the same time. Table 1 provides the formulas for

Table 3. Constant global memory for $m = 512$: wall clock time in seconds.

Nodes N	1	3	9	18	36
Processes p	8	24	72	144	288
$m = 512$	1.14	0.57	0.25	0.15	0.11

memory predictions for a global array of length n consisting of vectors of m doubles. The global vector is then divided into p local arrays of block length $l_n = n/p$ of m -vectors, such that the size of the local vectors is constant per number of processes p . The block size of the portions in the arrays *unsorted* and *sorted* that need to be communicated between process pairs is then the block size l_n/p of m doubles.

As explained in Section **Background**, we wish to use $N = 1, 3, 9, 18, 36$ nodes, which we choose by name so as to ensure their optimal connectivity in the leaf modules in the InfiniBand interconnect. Running then 8 processes on each node with two quad-core CPUs for most contention of network traffic, we have $p = 8N = 8, 24, 72, 144, 288$ parallel processes in a job. These numbers are listed in the first two rows of Table 1.

In all following experiments, we fix the length of the global array at $n = 2 \cdot (8 \cdot 18 \cdot 2 \cdot 3)^2 = 1,492,992$. This number is designed to ensure that all desired values of the block length l_n/p divide n without remainder. That is, n needs to be divisible without remainder not just by p , but by p^2 , since $l_n/p = n/p^2$. We had originally planned to consider also some other values of p , hence some additional factors are contained in the choice of n that are not strictly needed going forward; this generality does not impact the conclusions.

RESULTS

Experiment with Constant Global Memory

To effectively test the contention of the InfiniBand network, our team conducted a performance study with a constant global memory value, by fixing $m = 512$ as constant, which makes the global memory an estimated 6 GB for each of the two arrays *unsorted* and *sorted* in Table 2 for $n = 1,492,992$. The local memory of $l_n = n/p$ decreases from 729 MB to 20 MB, as the numbers of processes p and nodes N increase. This effect of decreasing memory is amplified for the block size l_n/p , namely from process to process it decreases by another factor of p , so that 93,312 kB decrease dramatically to 72 kB eventually.

Table 3 and Figure 2 both display the runtimes in seconds for the call to the `MPI_Alltoallv` command sending and receiving $m l_n/p$ doubles between processes for the choices of parameters in Table 2. The results show that the communication speed of the All-to-All command in fact decreases with additional nodes in the parallel job. The plot brings out how dramatic the decrease is.

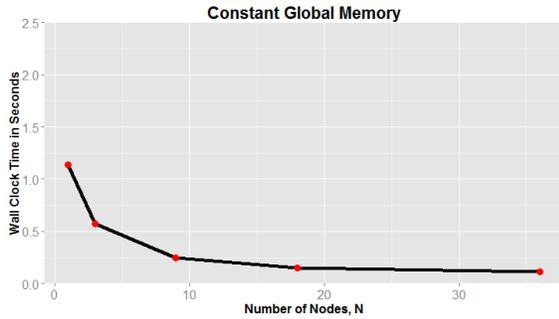


Figure 2. Constant global memory for $m = 512$: wall clock time in seconds vs. number of nodes.

Table 5. Constant local memory for $m = 512 N$: wall clock time in seconds.

Nodes N	1	3	9	18	36
Processes p	8	24	72	144	288
$m = 512 N$	1.14	1.64	2.09	2.28	2.30

This is remarkable and demonstrates that the high-performance InfiniBand interconnect can handle the stress of contention in All-to-All communications successfully, provided the overall problem does not increase in size, that is, for constant global memory. In the context of a larger algorithm that uses All-to-All communications, this communication will not be a bottleneck.

Experiment with Constant Local Memory

The results up to this point used a constant *global* memory with m constant for any number of p parallel processes, which leads to a rapidly decreasing block size l_n/p between pairs of processes. Now, in order to keep the block size in the All-to-All communications as large as possible, the vector length m will be designed to increase with increasing $p = 8N$, as reported in Table 4. The goal is to keep the block size l_n/p as large as possible, while p increases. This is limited by the requirement that the arrays *unsorted* and *sorted* need to fit in memory on each node. This implies that we cannot keep the block size l_n/p itself constant, but only the local memory controlled by l_n ; thus we pick the function $m = 512 N$, so that the local memory of each array *unsorted* and *sorted* is 729 MB for all values of p ; we call this the case of constant *local* memory. The block size l_n/p will then still decrease with increasing p , but less dramatically than before. This is seen in Table 4 in a decrease from 93,312 kB to 2,592 kB, which is a much larger final value than the 72 kB in Table 2. Notice the size of the global array increasing to a total of 205 GB on 36 nodes with increasing m , showing what significant problem size is eventually considered in this experiment.

The results displayed in Table 5 and Figure 3 present the runtimes in seconds, as we increase the number of processes $p = 8N$, while holding local memory on N nodes constant using $m = 512 N$. With the local memory held constant, the run times steadily increase as we increase

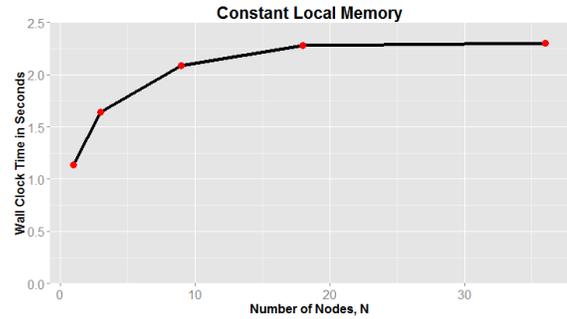


Figure 3. Constant local memory for $m = 512 N$: wall clock time in seconds vs. number of nodes.

N . The plot in Figure 3 brings the increase out very well, in particular compared to the decreasing line in Figure 2, which started from the same initial data point. Thus, in this case of maximum contention on the network, it is apparent that the run times increase with the numbers of processes, and the InfiniBand interconnect is eventually overcome by the All-to-All contention. For the use of All-to-All communication commands as building blocks in larger algorithms, this means that parallel scalability studies cannot succeed, since communication time worsens as the number of processes p increases.

CONCLUSIONS

As the results in Section Experiment with Constant Local Memory show, with local memory constant and contention on the network maximized, the run times for `MPI_Alltoallv` grow with the number of processes. This test case demonstrates that stress on the InfiniBand network can be created and will eventually limit the scalability of parallel algorithms that use All-to-All communications as building blocks. This is contrasted by the results in Experiment with Constant Global Memory that prove efficient behavior of the All-to-All communications, as long as the global memory stays constant, because this implies a dramatic decrease of the block size of the pairwise communications in the `MPI_Alltoallv` command. Both situations can occur in practice: The case of constant global memory occurs in strong scalability studies, where a fixed problem is divided onto the parallel processes, while the case of constant local memory appears in weak scalability studies, where the problem size is increased such that the local memory of each node is used at a constant level. Therefore, it depends on the structure of the code and its used, whether the InfiniBand interconnect becomes a limiting factor in parallel performance.

Acknowledgments

These results were obtained as part of the REU Site: Interdisciplinary Program in High Performance Computing (www.umbc.edu/hpcreu) in the Department of Mathematics and Statistics at the University of Maryland, Baltimore County (UMBC) in Summer 2013, where they

were originally reported in the tech. rep. [7]. This program is funded jointly by the National Science Foundation and the National Security Agency (NSF grant no. DMS-1156976), with additional support from UMBC, the Department of Mathematics and Statistics, the Center for Interdisciplinary Research and Consulting (CIRC), and the UMBC High Performance Computing Facility (HPCF). HPCF (www.umbc.edu/hpcf) is supported by the National Science Foundation through the MRI program (grant nos. CNS-0821258 and CNS-1228778) and the SCREMS program (grant no. DMS-0821311), with additional substantial support from UMBC. Co-author Jordan Ramsey was supported, in part, by the UMBC National Security Agency (NSA) Scholars Program through a contract with the NSA. Graduate RA Xuan Huang was supported by UMBC as HPCF RA.

REFERENCES

- Balaji, P., Buntinas, D., Goodell, D., Gropp, W., Kumar, S., Lusk, E., Thakur, R., and Träff, J. L. MPI on a million processors. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer, 2009, 20–30.
- Balaji, P., Narravula, S., Vaidyanathan, K., Krishnamoorthy, S., Wu, J., and Panda, D. K. Sockets direct protocol over InfiniBand in clusters: Is it beneficial? In *Performance Analysis of Systems and Software, 2004 IEEE International Symposium on-ISPASS*, IEEE (2004), 28–35.
- Hoeffler, T., Lumsdaine, A., and Rehm, W. Implementation and performance analysis of non-blocking collective operations for MPI. In *Supercomputing, 2007. SC'07. Proceedings of the 2007 ACM/IEEE Conference on*, IEEE (2007), 1–10.
- Kandalla, K., Mancini, E. P., Sur, S., and Panda, D. K. Designing power-aware collective communication algorithms for InfiniBand clusters. In *Parallel Processing (ICPP), 2010 39th International Conference on*, IEEE (2010), 218–227.
- Kandalla, K., Subramoni, H., Tomko, K., Pekurovsky, D., and Panda, D. A novel functional partitioning approach to design high-performance MPI-3 non-blocking Alltoallv collective on multi-core systems. In *42nd International Conference on Parallel Processing (ICPP) 2013*, IEEE (2013), 611–620.
- Kandalla, K., Subramoni, H., Tomko, K., Pekurovsky, D., Sur, S., and Panda, D. K. High-performance and scalable non-blocking all-to-all with collective offload on InfiniBand clusters: a study with parallel 3D FFT. *Computer Science-Research and Development* 26, 3-4 (2011), 237–246.
- Mistry, N., Ramsey, J., Wiley, B., Yanchuck, J., Huang, X., Gobbert, M. K., Mineo, C., and Mountain, D. Contention of communications in switched networks with applications to parallel sorting. Tech. Rep. HPCF-2013-13, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2013.
- Pacheco, P. S. *Parallel Programming with MPI*. Morgan Kaufmann, 1997.
- Raim, A. M., and Gobbert, M. K. Parallel performance studies for an elliptic test problem on the cluster tara. Tech. Rep. HPCF-2010-2, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2010.
- Reuther, A., Funk, A., Kepner, J., McCabe, A., Arcand, W., Currie, T., Hubbell, M., and Michaleas, P. Benchmarking the MIT LL HPCMP DHPI system. In *DoD High Performance Computing Modernization Program Users Group Conference, 2007*, IEEE (2007), 310–316.
- Thibeault, C. M., Minkovich, K., O'Brien, M. J., Harris Jr, F. C., and Srinivasa, N. Efficiently passing messages in distributed spiking neural network simulation. *Frontiers in Computational Neuroscience* 7 (2013).
- White, C. M. *Data Communications and Computer Networks: A Business User's Approach*. Course Technology, 2013.
- Wu, J., Wyckoff, P., and Panda, D. PVFS over InfiniBand: Design and performance evaluation. In *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, IEEE (2003), 125–132.

Table 1. Formulas for memory predictions.

Nodes N	1	3	9	18	36
Processes p	8	24	72	144	288
Dimension m	m	m	m	m	m
Length n of global array of m -vectors and their size in elements:					
Length n	n	n	n	n	n
Size	mn	mn	mn	mn	mn
Length $l_n = n/p$ of local arrays of m -vectors and their size in elements:					
Length l_n	$\frac{n}{p}$	$\frac{n}{p}$	$\frac{n}{p}$	$\frac{n}{p}$	$\frac{n}{p}$
Size	$m\frac{n}{p}$	$m\frac{n}{p}$	$m\frac{n}{p}$	$m\frac{n}{p}$	$m\frac{n}{p}$
Length l_n/p of block size of m -vectors in All-to-All and their size in elements:					
Length l_n/p	$\frac{n}{p^2}$	$\frac{n}{p^2}$	$\frac{n}{p^2}$	$\frac{n}{p^2}$	$\frac{n}{p^2}$
Size	$m\frac{n}{p^2}$	$m\frac{n}{p^2}$	$m\frac{n}{p^2}$	$m\frac{n}{p^2}$	$m\frac{n}{p^2}$

Table 2. Constant global memory for $m = 512$: predicted memory usage for one array.

Nodes N	1	3	9	18	36
Processes p	8	24	72	144	288
$m = 512$	512	512	512	512	512
Length n of global array of m -vectors and their memory in GB:					
Length n	1,492,992	1,492,992	1,492,992	1,492,992	1,492,992
Memory	6 GB				
Length $l_n = n/p$ of local arrays of m -vectors and their memory in MB:					
Length l_n	186,624	62,208	20,736	10,368	5,184
Memory	729 MB	243 MB	81 MB	41 MB	20 MB
Length l_n/p of block size of m -vectors in All-to-All and their memory in kB:					
Length l_n/p	23,328	2,592	288	72	18
Memory	93,312 kB	10,368 kB	1,152 kB	288 kB	72 kB

Table 4. Constant local memory for $m = 512N$: predicted memory usage for one array.

Nodes N	1	3	9	18	36
Processes p	8	24	72	144	288
$m = 512N$	512	1,536	4,608	9,216	18,432
Length n of global array of m -vectors and their memory in GB:					
Length n	1,492,992	1,492,992	1,492,992	1,492,992	1,492,992
Memory	6 GB	17 GB	51 GB	103 GB	205 GB
Length $l_n = n/p$ of local arrays of m -vectors and their memory in MB:					
Length l_n	186,624	62,208	20,736	10,368	5,184
Memory	729 MB				
Length l_n/p of block size of m -vectors in All-to-All and their memory in kB:					
Length l_n/p	23,328	2,592	288	72	18
Memory	93,312 kB	31,104 kB	10,368 kB	5,184 kB	2,592 kB

Parallel Performance of Higher-Order Methods on GPU Hardware

Tyler Spilhaus, Jared Buckley, and Gaurav Khanna
Physics Department, University of Massachusetts Dartmouth,
285 Old Westport Rd., North Dartmouth, MA 02747
gkhanna@umassd.edu

Keywords: GPGPU, Higher-Order, Scaling, OpenCL

Abstract

There is considerable current interest in higher-order methods and also large-scale parallel computing in nearly all areas of science and engineering. In this work, we take a number of basic finite-difference stencils that compute a numerical derivative to different orders of accuracy and carefully study the overall performance of each, on a many-core processor i.e. a graphics processing unit (GPU). We conclude that if one has a code that exhibits a high order of convergence, then there is likely to be only a modest gain through GPU parallelism in the context of total execution or “wall-clock” time. Conversely, for a low order code that exhibits good parallel performance, there is insignificant gain through the implementation of a higher-order convergent algorithm.

1. INTRODUCTION

In recent decades there has been a tremendous rise in numerical computer simulations, in nearly every area of science and engineering. In fact, computation has joined theory and experiment, as a third pillar in modern scientific research.

There is considerable current interest in harnessing the advancements made in video gaming technology for scientific high-performance computing (HPC). An example of this trend is the rapid rise in the use of custom-designed HPC graphics cards (GPUs) as “accelerators” in workstations and even large supercomputers. In fact, the second-fastest supercomputer today, ORNL’s *Titan*, makes use of Nvidia’s custom-HPC Kepler GPUs to achieve *petascale* performance [1]. And there are around 50 such GPU-accelerated systems in the top 500 supercomputers worldwide. One reason for this recent trend is that the massive consumer video-gaming market significantly helps in “subsidizing” the R&D cost associated towards advancing these compute technologies resulting in high performance at a low cost. In addition to their cost effectiveness, these GPU technologies are substantially “greener” over CPUs in the sense of the high performance delivered per Watt of electrical-power consumed [2].

In addition to the strong increase in the interest in many-core GPU computing, there is also a rising trend in developing numerical algorithms that converge faster than the common second-order accurate schemes [3]. Some examples of such higher-order convergent methods are – higher-order finite-differencing, spectral collocation method, radial basis function method, finite-element and others [4–12]. In this work, we restrict ourselves to higher-order finite-difference schemes, however, we anticipate that our findings are generic enough that they would apply to any higher-order method.

The main goal of this work is to clearly demonstrate a form of “trade-off” between parallel computing and higher-order methods. This trade-off stems from the detailed parallel performance of the various higher-order schemes under specific conditions. In particular, our main assumption in this work is that the physical or engineering problem to be solved numerically has a known degree of tolerance or error acceptable for the solution, and is a given fixed quantity. We interpret this error to be the scale of the “discretization” or truncation error arising from the numerical scheme, which is, of course, a significant simplification – however, one that is reasonable for a wide class of problems. In other words, *we study the parallel performance of different order finite-difference methods given a fixed level of the discretization error.*

The outcome of our study suggests several very significant conclusions: (a) If one has a parallel code that performs well and exhibits second-order convergence, there is no gain to be expected from a higher-order method implementation, assuming one has a large enough computational resource available, and the major consideration is the total execution time; (b) if one has a serial code exhibiting higher-order convergence (say, higher than fourth-order) then there is only a modest gain from a parallel algorithm in a similar context; and (c) depending upon the acceptable error level, there is likely an optimal approach i.e. a combination of parallelism and method-order that would be ideal for the problem.

This article is organized as follows: In Section 2, we present a simplified description of a many-core GPU and express the expected outcome of our planned study based on simple heuristic reasoning. In Section 3, we detail the method of our study and present explicit mathematical expressions and our approach towards GPU parallelism. In Section 4, we show and discuss our results, and we end with some conclu-

sive remarks in Section 5.

2. OPENCL & MANY-CORE GPU ARCHITECTURE

We make use the Open Computing Language (OpenCL) [13] parallel programming framework in this work. OpenCL is a framework for programming across a wide variety of computer hardware architectures (CPU, GPU, Heterogeneous, DSP, etc). In essence, OpenCL incorporates the changes necessary to the programming language C, that allow for parallel computing on all these different processor architectures. In addition, it establishes numerical precision requirements to provide mathematical consistency across the different hardware and vendors – a matter that is of significant importance to the scientific computing community. Computational scientists would need to rewrite the performance intensive routines in their codes as OpenCL *kernels* that would be executed on the compute hardware. The OpenCL API provides the programmer various functions from locating the OpenCL enabled hardware on a system to compiling, submitting, queuing and synchronizing the compute kernels on the hardware. Finally, it is the OpenCL runtime that actually executes the kernels and manages the needed data transfers in an efficient manner. Most vendors have released an OpenCL implementation for their own hardware.

In OpenCL, the GPU (called *device*) is accessible to the CPU (called *host*) as a co-processor with its own memory. The device executes a function (usually referred to as a *kernel*) in a data-parallel model, which means that a number of threads run the same program on different data. A *warp* is a group of 32 threads, which is the smallest grouping of threads that is processed by a GPU compute-unit. In OpenCL, work is performed with in groups of threads that are assembled into a *workgroup*. If the GPU has a large number of compute-units, it can process them in parallel. Thus, in the context of high performance computing using a GPU, massive parallelism is extremely important. The architecture allows one to farm out a large number of calculations into a series of groups and execute them in parallel. The threads then scatter, doing all of their farmed computations on the different compute-units, and synchronize at the completion of their assigned tasks.

Now, we present a simplified argument that will help argue our main claim and may provide some predictive value for other codes beyond the simple sample code we consider in this work.

Let us say that one is interested in performing a simple one-dimensional (1D) numerical derivative using a second-order finite-difference stencil. An important parameter that must be chosen is the grid resolution, that is typically set by the grid size N . The number of numerical calculations necessary to perform the derivative computation on the entire grid would then be on the order of N . Now, let’s assume that one

attempts the same computation on a GPU with with a large number of compute-units. Given the massively-parallel design of the GPU architecture (as explained above), we expect that as N increases, the computation scales up to the optimal capacity of the GPU and therefore exhibits near perfect *weak* scaling. Thus, the “wall-clock” time would stay nearly constant even though N is increasing. This would continue to point, of course, and eventually an increase in N further, would result in a corresponding increase in wall-clock time.

Now, let us consider the same in the context of a higher-order finite-difference stencil of order p in 1D. For the *same* level of error as the second-order case above, one would only need a grid size on the scale of $N^{2/p}$ which is significantly smaller. Of course, one would expect near perfect weak scaling in this case as well, and therefore the wall-clock time would remain constant as N increases. However, each compute-unit of the GPU would need to perform additional computational work in order to execute the more complex higher-order algorithm. In fact, the wider higher-order stencil requires access to additional grid points, and more numerical computation as compared with the second-order case. Crudely, the work performed by each compute-unit would increase by a factor of $p/2$ over the second-order case. This implies that *the GPU performance of higher-order methods, in general, is expected to be worse compared to the second-order case, in the context of a fixed discretization error.*

The question that we will address in this work, is whether the improved parallel performance of lower order methods is *enough* to give them an advantage in the context of the most important aspect of a numerical computation – the *total execution or wall-clock time*. This is the main point that we explicitly investigate in the following sections using finite-difference schemes of order 2, 4, 6 and 8.

3. METHODOLOGY

In this section, we describe in detail the method of study adopted in this work. We begin with a discussion of higher-order finite-difference stencils, followed by our results from correlating the error with grid size N and end with a description of our parallel GPU implementation.

For the finite-difference calculations, we used a cosine function on a 1D domain from 0 to 12π :

$$f(x) = \cos x, \quad x \in [0, 12\pi]$$

Now, for a numerical implementation, x is discretized simply as:

$$x_i = ih$$

where

$$h = \frac{12\pi}{N}$$

and i is an index that labels an arbitrary grid point on the domain.

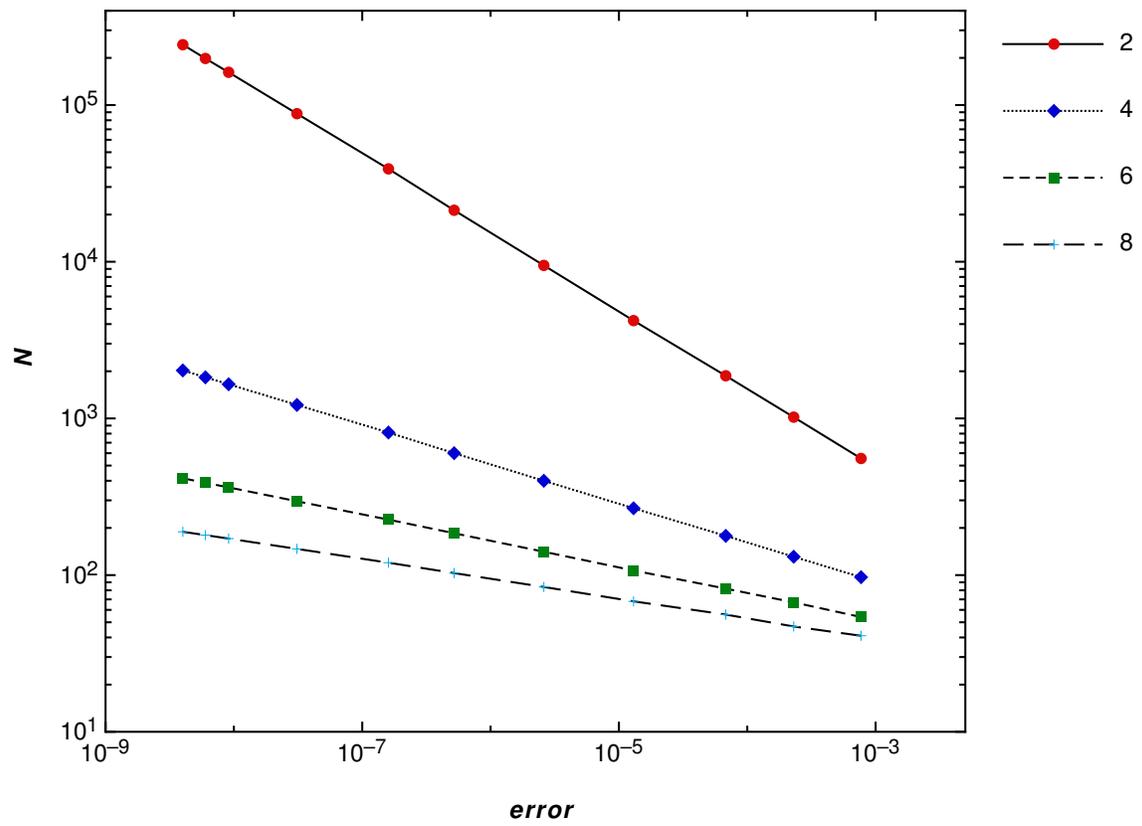


Figure 1. The correlation of N with the error for various finite-difference orders. The powers of N computed from this data are: -1.998 , -3.955 , -5.942 and -7.876 respectively.

To calculate the derivative of a function at grid point i using the finite-difference schemes, it is necessary to use the function values at neighboring grid points. As the order of the scheme increases, more grid points are needed for the calculation.

In the context of this work, we focus our attention on the first derivative of $f(x)$. At grid point i , using the various different order central finite-difference schemes, the derivative is given as [3]:

Order 2:

$$\frac{1}{h} \left(-\frac{1}{2} f_{i-1} + \frac{1}{2} f_{i+1} \right)$$

Order 4:

$$\frac{1}{h} \left(\frac{1}{12} f_{i-2} + \frac{-2}{3} f_{i-1} + \frac{2}{3} f_{i+1} + \frac{-1}{12} f_{i+2} \right)$$

Order 6:

$$\frac{1}{h} \left(\frac{-1}{60} f_{i-3} + \frac{3}{20} f_{i-2} + \frac{-3}{4} f_{i-1} + \frac{3}{4} f_{i+1} + \frac{-3}{20} f_{i+2} + \frac{1}{60} f_{i+3} \right)$$

Order 8:

$$\frac{1}{h} \left(\frac{1}{280} f_{i-4} + \frac{-4}{105} f_{i-3} + \frac{1}{5} f_{i-2} + \frac{-4}{5} f_{i-1} + \frac{4}{5} f_{i+1} + \frac{-1}{5} f_{i+2} + \frac{4}{105} f_{i+3} + \frac{-1}{280} f_{i+4} \right)$$

where the notation, $f_i = f(x_i)$.

Because we are concerned with the scaling at a fixed level of error, it was first necessary to correlate N with a given error level. This was achieved using an iterative algorithm that searched for a given error value for each finite-difference order being investigated. The algorithm calculated the first derivative of the cosine function in two ways: using the math library sine function and using the finite-difference formula at a value of N . The error was calculated at each point in the domain, and the maximum error on the domain was compared to the given error value. If the calculated error was less than the given error, the value of N was recorded; otherwise, N was incremented up by one and the process was repeated. The correlation of N with the error for each investigated finite-difference order is given in Tab. 1 and Fig. 2. The values of N fit the expected patterns extremely well.

Now, on the GPU, we implemented these different order calculations as separate OpenCL kernels. Each kernel thread computes the derivative at one grid point using the appropriate finite-difference stencil above. We ran each kernel several million times and carefully timed the total runtime. Several workgroup sizes were tried in each case and the optimal values were used for the final benchmark.

Our test GPU in this study was an *AMD Radeon HD R9 290x* – one of the best performing consumer-grade GPUs available to-date. This device has 2816 compute-elements, 4GB of GDDR5 memory and 512-bit memory-bus for industry-leading memory bandwidth. The compute-elements are clocked at a frequency of 1 GHz.

Table 1. Correlation of N and Error

Error	2	4	6	8
7.7×10^{-4}	555	97	54	41
2.3×10^{-4}	1019	131	67	47
6.8×10^{-5}	1871	178	82	56
1.3×10^{-5}	4210	267	107	68
2.6×10^{-6}	9473	400	141	84
5.2×10^{-7}	21313	600	185	103
1.6×10^{-7}	39153	813	226	120
3.1×10^{-8}	88094	1219	296	147
9.1×10^{-9}	161843	1652	363	171
6.0×10^{-9}	198222	1829	388	180
4.0×10^{-9}	242766	2024	415	189

4. RESULTS

In this section we present the results obtained due to the approach and methodology as detailed in the previous sections.

In Fig. 2 we depict the speedup as a function of the local workgroup size, for a grid size $N = 39, 153$ or error-level 1.5×10^{-7} . The speedup is defined relative to the second-order computation utilizing a single thread per workgroup. We show the same for all the method orders considered in this work, namely 2, 4, 6 and 8. The expectations, as presented in Section 2, are clearly borne out in our speedup data. The second-order method, scales the best, and the higher-order cases exhibit much poorer scaling. In fact, the sixth and eight-order cases barely scale at all! One is clearly better off running those simply in serial mode.

In Fig. 3 we show the speedup as a function of the error level for multiple order methods. The *speedup is defined relative to the second-order computation using a single thread per workgroup, and it is obtained by choosing the value of the workgroup size for the least wall-clock time*. It is clear that the second-order method performs the best for a majority of the cases presented, and quantitatively agrees with our expectation of a factor of $p/2$ speedup over a method of order p . As argued in the previous section, this is due to the fact that the second-order method has a simpler finite-difference stencil and exhibits near perfect weak scaling on the GPU architecture. As one goes from right to left decreasing the error level, at some point the fourth-order begins to perform better due to the rather rapid increase in N in the second-order case. This happens for error levels that are lower than one part

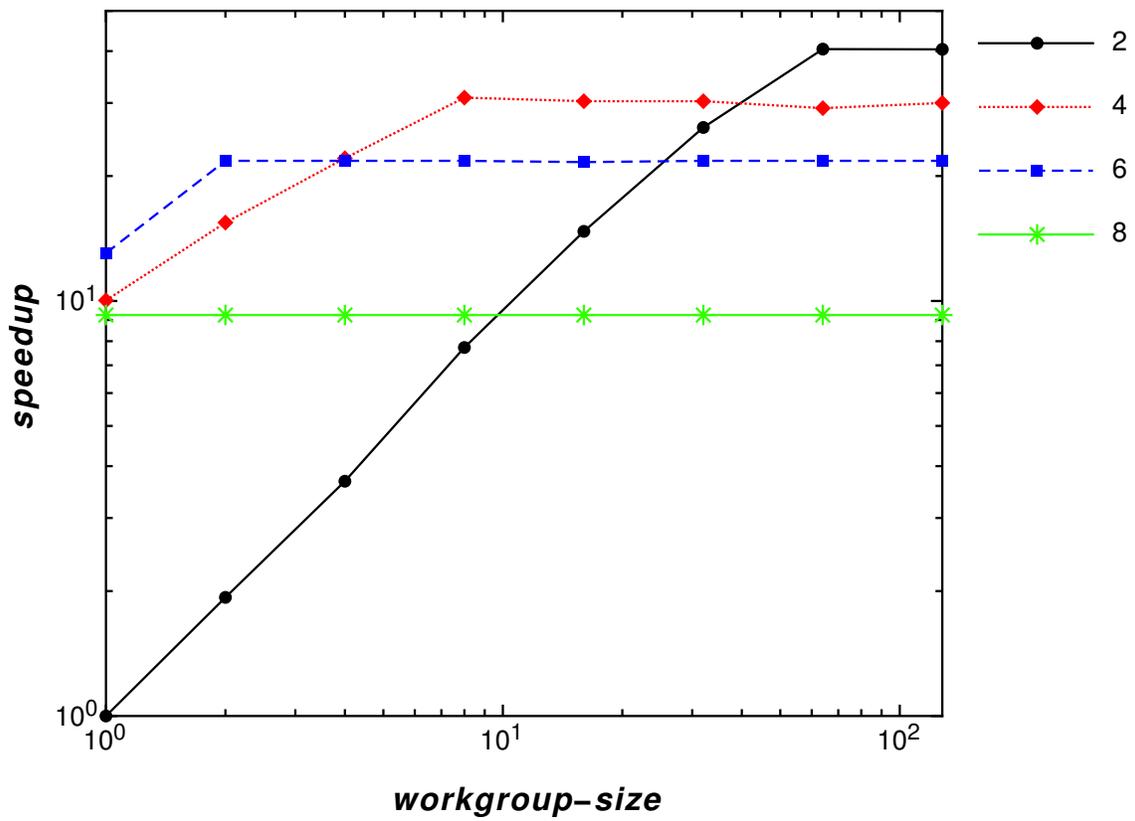


Figure 2. Speedup as a function of the workgroup size for a fixed level of error, for various finite-difference orders. Speedup is defined relative to the second-order calculation running using a single thread per block. It is clear that the second-order method scales very well comparatively.

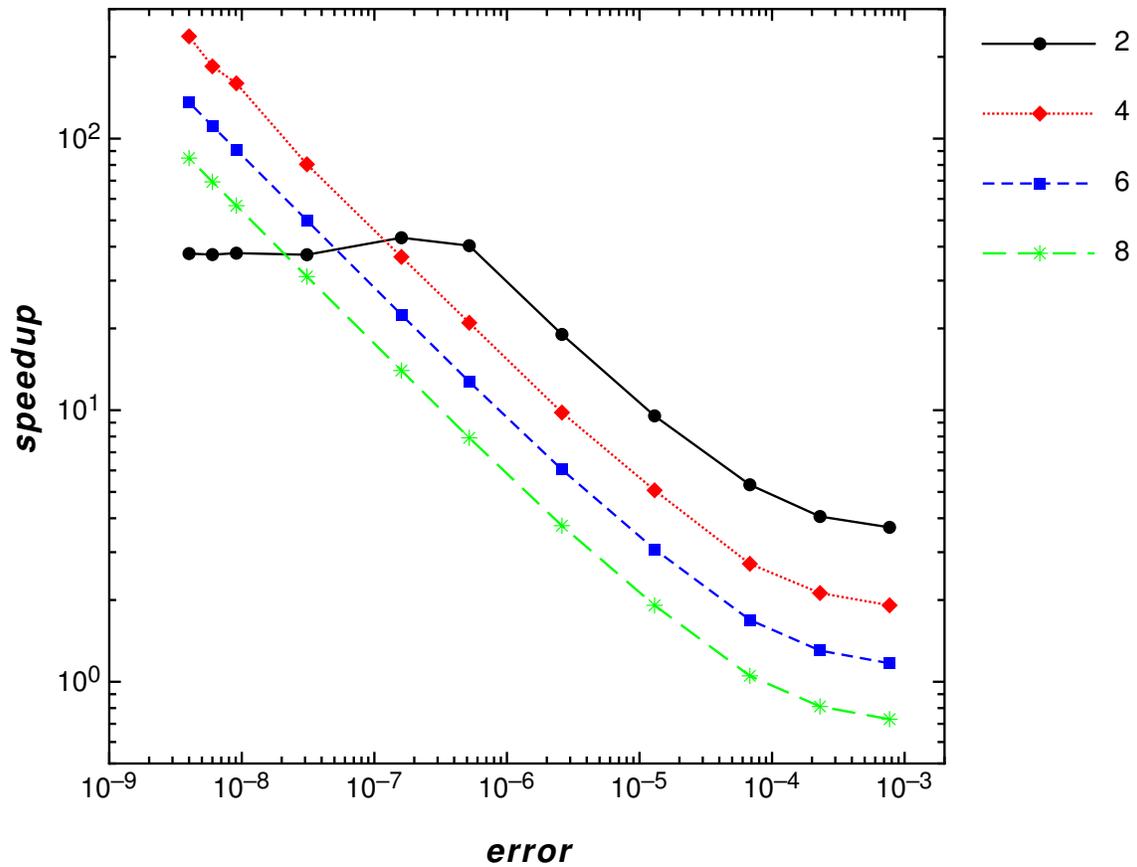


Figure 3. Best performance (speedup) for various finite-difference order methods on a many-core GPU. Speedup is defined relative to the second-order computation using a single thread per workgroup. It is clear that the second-order method performs the best overall for most practical levels of error.

in a *million*. Very few science and engineering problems are likely to require error levels that low. It's interesting that in our study we found no evidence of any benefit arising from methods higher than fourth-order.

In summary, our expectations as argued in Section 2 are completely borne out in our results as presented here. For most practical cases of interest, the second-order method performs the best on a many-core GPU, by a single-digit factor.

5. CONCLUSIONS

In this work, we have demonstrated that different order finite-difference methods exhibit different overall performance on a parallel many-core GPU. In general, given a fixed level of accuracy, lower-order methods perform better due to the fact that they have a simpler stencil and the larger grid sizes scale well onto the GPU architecture. Using a basic example, we have been able to show that *the gain in performance from improved parallel performance of the second-order method is enough to have its overall performance exceed that of a higher-order method by a factor of a few*.

Conversely, we have been able to show that a *higher-order method, say, eighth-order, converges so well, even on a modest grid size that such a method simply does not require any parallel resources at all*. While we have made our arguments and claims using a simple 1D derivative finite-difference stencil, we expect our main outcomes to hold more generally, including even in 2D and 3D. In addition, it should be noted that although we worked with a simple numerical derivative model for pedagogical reasons, the main outcomes mentioned here apply in a much broader context i.e. in the context of any higher-order convergent algorithm that is implemented for execution in parallel environment.

To conclude, with the interest of minimizing total execution time and given a sufficiently large computational resource, a "brute force" approach with a lower-order method is likely to outperform a more advanced, highly convergent, higher-order method. Since many "real world" science and engineering research codes are written using second-order accurate algorithms, and it is often very challenging to develop algorithms that converge faster, an investment in GPU code development may prove to be quite worthwhile. We believe that developing and implementing a parallel algorithm that scales reasonably well, is easier than one that exhibits faster convergence. This is especially true for sophisticated research codes. On the other hand, if one already has a fast converging serial code then it may be much better to invest in making modest improvements to the convergence rate as opposed to a fully parallel implementation.

Finally, a subset of the authors have also performed a related study i.e. scaling performance of higher-order methods in a distributed parallel environment (a computer cluster) [14]. The results obtained therein are well aligned with the

outcome of this present work.

6. ACKNOWLEDGEMENTS

The authors acknowledge research support from NSF Grants No. PHY-1016906, No. CNS-0959382, No. PHY-1135664, No. PHY-1303724, and No. PHY-1414440, and also from the U.S. Air Force Grant No. FA9550-10-1-0354 and No. 10-RI-CRADA-09.

7. REFERENCES

- [1] The Top 500 List: <http://top500.org/>
- [2] The Green 500 List: <http://green500.org/>
- [3] B. Gustafsson, "High Order Difference Methods for Time Dependent PDE", Springer Series in Computational Mathematics, Volume 38 (2008).
- [4] J.S Hesthaven, T. Warburton, "Nodal High-Order Methods on Unstructured Grids: I. Time-Domain Solution of Maxwell's Equations", Journal of Computational Physics, Volume 181, Pages 186-221 (2002).
- [5] M. O. Deville, P. F. Fischer, E. H. Mund, "High-Order Methods for Incompressible Fluid Flow", Cambridge University Press (2002).
- [6] G. Karniadakis, "High-order splitting methods for the incompressible Navier-Stokes equations", Journal of Computational Physics, Volume 97, Pages 414-443 (1991).
- [7] D. Xiu, J. S. Hesthaven, "High-Order Collocation Methods for Differential Equations with Random Inputs", SIAM J. Sci. Comput., 27(3), 1118-1139 (2005).
- [8] C-W. Shu, "High-order Finite Difference and Finite Volume WENO Schemes and Discontinuous Galerkin Methods for CFD", International Journal of Computational Fluid Dynamics, Volume 17, Issue 2, Pages 107-118 (2003).
- [9] J.T. Beale, "High order accurate vortex methods with explicit velocity kernels", Journal of Computational Physics, Volume 58, Pages 188-208 (1985).
- [10] M. R. Visbal, D. V. Gaitonde, "High-Order-Accurate Methods for Complex Unsteady Subsonic Flows", AIAA Journal, Volume 37, No. 10, pp. 1231-1239 (1999).
- [11] J. Shen, T. Tang, "Spectral and High-Order Methods with Applications", Mathematics Monograph Series 3 Science Press (2006).
- [12] J. Hesthaven, D. Gottlieb, S. Gottlieb, "Spectral Methods for Time-Dependent Problems", Cambridge Monographs on Applied and Computational Mathematics (2007).
- [13] OpenCL website: <https://www.khronos.org/opencl/>
- [14] J. Buckley, G. Khanna, "Parallel Scaling Performance and Higher-Order Methods", Proceedings of the WORLD-COMP14 Conference (CSC), Las Vegas, Nevada (2014).

DOEE: Dynamic Optimization framework for better Energy Efficiency

Jawad Haj-Yihia
CPU Architecture
Intel Corporation
jawad.haj-yihia@intel.com

Ahmad Yasin
CPU Architecture
Intel Corporation
ahmad.yasin@intel.com

Yosi Ben-Asher
Computer Science Department
University of Haifa
yosi@cs.haifa.ac.il

ABSTRACT

The growing adoption of mobile devices powered by batteries along with the high power costs in datacenters raise the need for energy efficient computing. Dynamic Voltage and Frequency Scaling is often used by the operating system to balance power-performance. However, optimizing for energy-efficiency faces multiple challenges such as when dealing with non-steady state workloads. In this work we develop DOEE - a novel method that optimizes certain processor features for energy efficiency using user-supplied metrics. The optimization is dynamic, taking into account the runtime characteristics of the workload and the platform. The method instruments monitoring code to search for per-program-phase optimal feature-configurations that ultimately improve system energy efficiency. We demonstrate the framework using the LLVM compiler when tuning the Turbo Boost feature on modern Intel Core processors. Our implementation improves energy efficiency by up to 23% on SPEC CPU2006 benchmarks, outperforming the energy-efficient firmware algorithm. This framework paves the way for auto-tuning additional CPU features.

Author Keywords

Compiler, DVFS, Energy Efficiency, Power, Dynamic Optimizations.

ACM Classification Keywords

D.3.4 PROCESSORS

INTRODUCTION

Energy efficiency has become one of the most important design parameters for hardware, due to battery life on mobile devices and energy costs and power provisioning in data-centers. Performance features like Dynamic Voltage and Frequency Scaling (DVFS), Turbo Boost or memory prefetching are offered by hardware manufacturers for

HPC 2015, April 12 - 15, 2015, Alexandria, VA, USA

© 2015 Society for Modeling & Simulation International (SCS)

software use. However, utilizing such features is tricky as it comes with a power cost. Completely disabling these features often incur significant slowdowns that may also waste battery budget. In some scenarios performance is favored over power (e.g. responsiveness in smartphones). In other scenarios, a good-enough performance can be quite effective at sustaining a long battery life (e.g. video playback).

In this work we propose a novel *framework* called DOEE (Dynamic Optimization for Energy Efficiency) that optimizes the system energy efficiency. A *dynamic* approach is adopted where CPU energy telemetry and performance counters are periodically sampled. The framework is *adaptive* where it searches for an optimal point meeting a *user-supplied* metric for energy-efficiency. Metrics like energy, performance, or combinations of them are illustrated in section 2.

We *demonstrate* the framework using the Turbo Boost [1] feature available on modern Intel processors. The framework spares the power budget in order to enable Turbo in potential phases in more energy efficient manner, resulting in reduced energy with nearly the same or better performance. Our results outperform the energy efficiency algorithm implemented by the firmware of Intel processors. To the best of our knowledge, this is the first work that attempts to tune Turbo Boost.

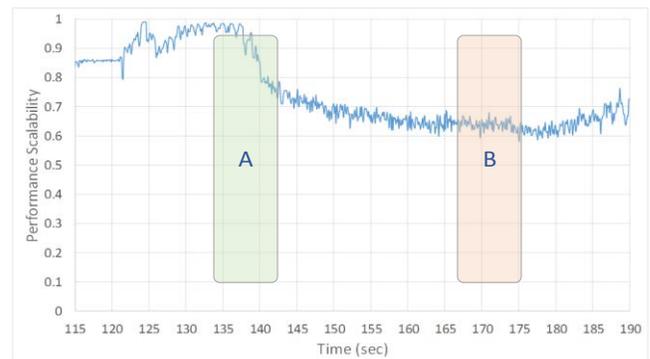


Figure 1: Performance Scalability over time for 429.mcf on Haswell machine (seconds 115 to 190)

To illustrate the idea, Figure 1 shows the Performance scalability over time of a sample application running with fixed frequency. Consider the phases A and B with average scalability of 97% and 62% respectively (both are 10

seconds long). Scalability is performance and frequency correlation; a ratio of 1 means doubling frequency results in doubling performance.

Table 1: Energy-efficiency comparison of the two phases

Phase	Average Scalability	Delay (sec)	Energy (J)	EDP	ED ² P
A	97%	7.8	140.7	1106.8	8704.31
B	62%	8.5	150.4	1282.3	10926.7
Gain at A over B		7.7%	6.4%	13.6%	20.3%

Table 1 describes the energy efficiency differences by applying the Turbo at A or B. We can see that by applying the Turbo at phase A instead of applying it at phase B, the overall run time was reduced by 7.7% and an energy gain of 6.4% is achieved. Other metrics should even bigger improvements.

The main contributions of our work:

- A novel framework is developed for Dynamic Optimization for Energy-Efficiency: DOEE is simple (no prior calibration is required), configurable (user metric) and adaptive (per dynamic characteristics).
- The DOEE framework is demonstrated by tuning the Turbo feature using the LLVM compiler. Our solution outperforms IvyBridge processor built-in algorithm.
- Our framework/implementation is made available for the research community [26]. Additionally, a few architectural enhancements are proposed to aid such approaches.

BACKGROUND

Turbo

Turbo Boost [1] – also known as Intel Turbo Boost Technology 2.0 introduced with Intel’s 2nd generation Core™ processor – opportunistically boosts the frequencies of the cores in multi-core Intel Processors. The processor hardware controls the Turbo Boost activation and the level of boosting depends on the number of active cores, estimated power consumption, and the temperature of the package. This “thermal boosting” allows the processor to temporarily exceed the Thermal Design Point (TDP) using the thermal capacitance of the package. Figure 2 [12] illustrates the Turbo behaviors over time. In the first phase (sleep or Low power), the processor gains thermal budget while sleeping or running at low-power. In the second phase (C0/P0) the processor moves to Turbo (P0 in the P-state terminology [17]) following a P0 request by the software. In this stage the processor starts to consume its thermal budget and at a later stage as the processor heats up the processor’s hardware starts to reduce the frequency as the its temperature gets close to the maximum allowed temperature. Once the entire thermal budget is consumed the processor’s frequency normally stabilizes at the

frequency corresponding to TDP frequency. The processor is not allowed to go back to Turbo until a new thermal budget is accumulated.

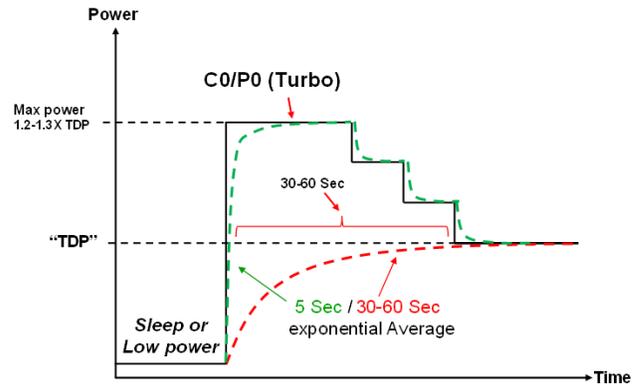


Figure 2: Turbo Boost behavior overtime [12], thermal budget gained at low-power intervals is used to temporarily run above TDP frequency

Turbo Boost might be energy inefficient in some cases. To cope with this problem, Intel’s modern processors feature the ability for Software to control the energy efficiency of the processor by configuring the IA32_ENERGY_PERF_BIAS register [17]. The processor can be configured to favor highest performance, maximum energy savings or a value in-between. Notice the energy efficiency problem is not straightforward to be solved by hardware alone. Thus as software-assisted approach is used to give hints to the underlying hardware about the software preferences to better handle energy efficiency. For example, different vendors might prefer different efficiency metrics. Commonly used ones are illustrated in the remainder of this section.

Energy Efficiency Metrics

The metric of interest in power studies varies depending on the goals of the work and the type of platform being studied. In some situations, focusing solely on energy is not enough. For example, reducing energy at the expense of lower performance may often not be acceptable. On the other hand, gaining performance at the expense of high energy consumption might not be practical for the system under design. Thus, metrics combining energy and performance have been proposed. Here we give a short survey of the various metrics used:

Energy - This metric is important in mobile systems. The unit of energy is Joules. Energy usage, which is closely correlated to battery life and battery capacity, is usually measured in watt-hours (Wh) which is an energy unit (1Wh=3600 Joules). Energy is also important in non-mobile platforms. For data centers [6], energy consumption is one of the leading operating costs (electricity bills), and thus reducing the energy usage is critical in these systems.

Power - is the rate at which energy is consumed. The unit of power is watts (W), which is Joules per second. This metric

is important for designing the power-delivery network (current and voltage requirements). In addition this helps in understanding the power density of the system, which is used for thermal studies in the process of building a cost-efficient cooling solution.

Energy-delay product (EDP) - is a metric that was proposed [8] to take into account the energy and performance at one metric. If either energy or delay increase, the EDP will increase. Thus, lower EDP values are desirable. EDP's inclusion of runtime means that this is a metric that improves with approaches that either hold energy constant but execute the same instruction mix faster, or hold performance constant but execute at a lower energy, or some combination of the two.

Energy-delay-squared product (ED²P) [9, 10,11] - is similar to EDP but gives more weight to performance (1/delay) than energy cost.

METHODOLOGY

To improve the energy efficiency we used an automatic search method for finding the optimum configuration for a given energy efficiency metric. For each program phase (loop or function) we do a competition, traversing possible configurations of the feature, subject to dynamic tuning. The algorithm is described in Figure 3, we start the search from the highest allowed frequency (P0) down to the lowest allowed frequency (Pn), and stop the search once we reach a frequency that minimize the energy-efficiency metric. The metric itself is configurable by the user – such as Energy or EDP discussed in previous section. The next runs of that program phase will use the best-performing configuration found by the search (competition) phase, optimal frequency in our example.

The instrumented code by the framework can be divided into a few stages. Stage I is the *search stage* (light blue color), where we try various configurations (e.g. various frequencies) and we choose the optimal one for the particular program phase. In stage II we apply the winning configuration chosen in stage I to future runs of the same program phase (green color). As the optimal frequency might change due to changes at the platform (e.g. number of cores running, Display or Imaging state, communication traffic [22]), there is a periodic re-search (waiting in stage III in orange color), after a periodic time we trigger the search stage again to capture system level changes.

The algorithm is applied to program phases, which generally start at the function's entry and loop's entry (pre-header of the loop). These points are the start of potential program phases. At the first inspection of the potential phase we capture the energy accumulator counter (RAPL[17]) and the Time-Stamp-Counter (TSC). At the next inspection we capture the same counters and calculate the delta from previous sample. Figure 4 describes the entry point location in case of loop or function. Both counter value captures are done at the entry point to the loop or the function, as this allows for the capturing of the execution of

serial loops as shown in Figure 4.a. Moreover, recursive function calls will be captured at the entry of the function shown in Figure 4.b. To minimize the number of configuration change overhead, short (below some instruction threshold e.g. 100 thousand instruction) function and loops were skipped. In such case instructions counter is not reset at the exit from the loop or function, but rather resume counting until we return back to the entry-point of the specific function or loop. This method allows capturing long chains of short functions/loops that are repeatedly executed (e.g. in Figure 4.a: Inner_A → Inner_B → Inner_C → Inner_A).

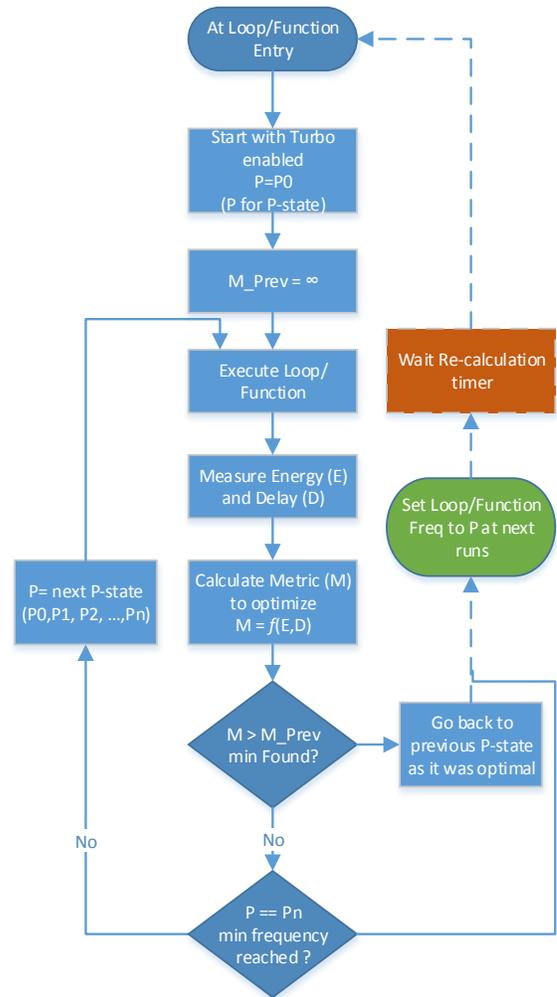


Figure 3: Auto-tuning algorithm to find frequency point (P) that minimizes the evaluated metric (M)

IMPLEMENTATION

Our framework consists of the LLVM (Low Level Virtual Machine) compiler [18], performance counters out of the Performance Monitoring Units (PMU) [17], DVFS, energy telemetry (Intel RAPL [17]). We've modeled the Auto-

Tuning algorithm that uses the data to choose more energy-efficient work. In addition we wrote a kernel-mode driver module to configure the Performance Counters at startup. The performance counters and TSC are queried from user land using the RDPMC instruction in order to limit the runtime overhead. The Framework is described in Figure 5.

The program is being compiled with our modified LLVM compiler. The compiler instruments Auto-tuning code that implements our algorithm from Figure 3, and outputs assembly code for the target machine.

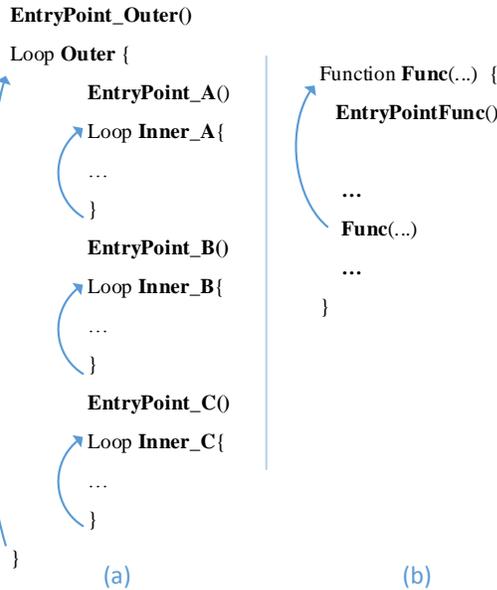


Figure 4: Entry Points to loops and functions where the auto-tuning algorithm is invoked

DVFS

In the Intel processors, DVFS is controlled by choosing Performance state (P-state), this is done by writing to per-thread Model-Specific Register (MSR) accessed by the kernel. The power-management unit considers the requests from all running threads and sets the frequency to the maximum frequency requested by all of the threads. The requested P-states are values between P0 to Pn, where P0 is the maximum frequency with Turbo enabled, P1 is the maximum guaranteed frequency and Pn is the lowest available frequency. In our system the P0 is 2.8GHz, P1 is 1.9GHz and Pn is 0.8GHz. The DVFS request and the Turbo control is done by writing to the IA32_PERF_CTL MSR [17].

Energy measurements

Intel introduced the Running Average Power Limit (RAPL) feature with the Sandy Bridge microarchitecture [12].

These energy counters [17] give visibility to the energy consumption of the package, cores, graphics (in client processor) and memory domains (in server processor). Today there is no option to read the whole platform energy consumption (energy consumed from the battery or power source); we expect that this option will be available in the near future by the processors vendors.

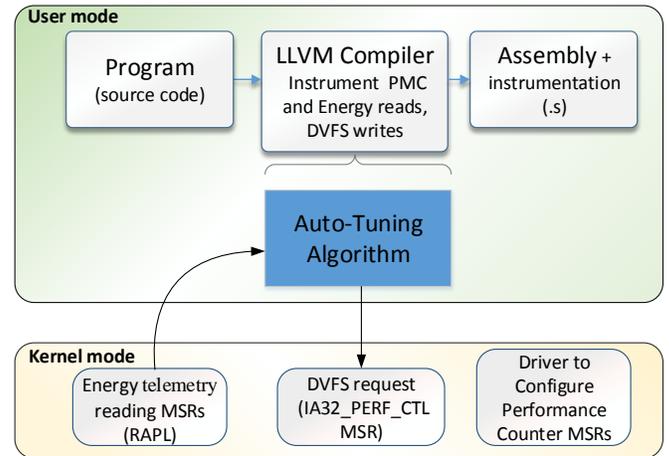


Figure 5: Block-diagram of main framework's components

RAPL data can be configured and examined by reading MSRs. On the Intel architecture, today this is only possible in privileged kernel mode.

The available RAPL energy counter for cores domain is for all the cores together. Several studies have been done for CPU performance modeling per-core and per-thread energy using performance counter [13-16]. For our study we are using the PKG_ENERGY_STATUS MSR that reflects the energy consumed by the whole CPU package.

Code instrumentation

The auto-tuning code is instrumented at the beginning of program phases (functions and loops), and in addition the compiler adds global variables that enable tracking the frequency changes by programs phases (e.g. not setting the frequency to the same value twice).

Architectural enhancements

We propose the following enhancements to aid DOEE-based optimizations and reduce overheads of our method:

- Add new user level DVFS instruction to reduce the overhead of our method.
- Faster DVFS transition implementation. We believe that this is feasible with the new technologies of on-die voltage regulators [21].
- Low latency RDPMC and RDMSR instructions, especially reading the RAPL energy MSRs.
- Higher resolution energy reporting. In our study, RAPL counters are updated nearly every one millisecond. More frequent updates enable finer-grain optimization.

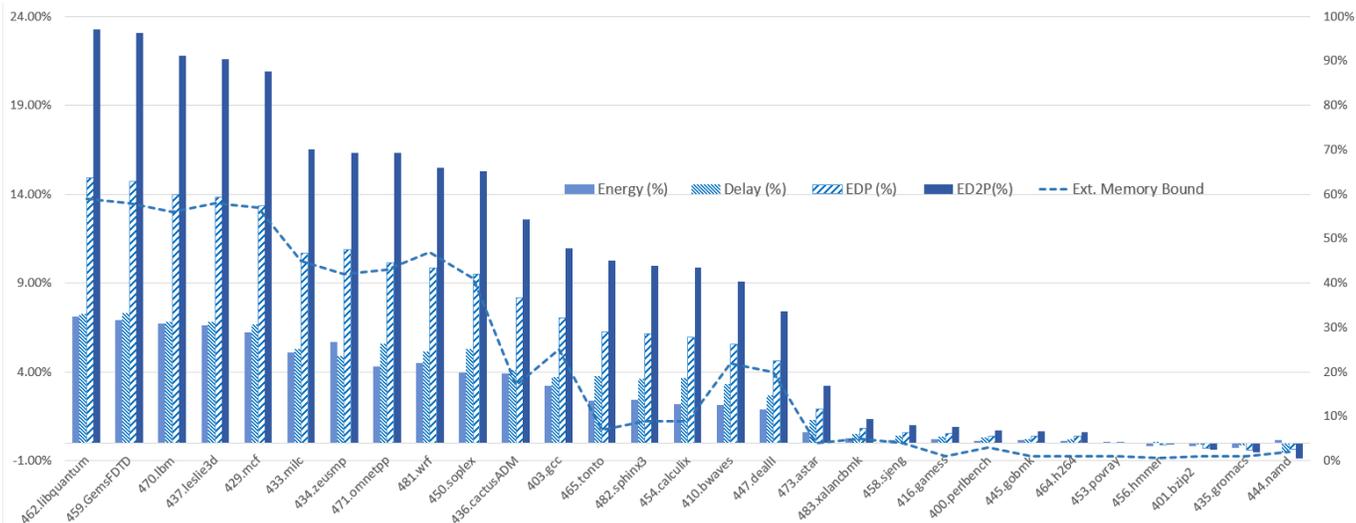


Figure 6. Auto-Tuning Algorithm measured gains on CPU2006 workloads for commonly used metrics

RESULTS

The implementation was tested on a 3rd Generation Intel® Core™ i7 3517U Processor code name Ivybridge. SPEC CPU2006 benchmarks [20] are measured in rate single-copy configuration using the reference input sets. Compiler flags used (-O3). For benchmarks with more than one input we used the first one. The IA32_ENERGY_PERF_BIAS MSR was set to 7. A value of 7 translates into a hint to the processor to balance performance with energy consumption while a value of 0 or 15 translated to highest performance and highest energy savings respectively [17].

Figure 6 shows the results of the runs to improve the Turbo energy-efficiency, four metrics are shown: Energy, Delay, EDP and ED²P. In addition the graph shows the average Ext. Memory Bound[27] for each benchmark. This metric represents the fraction of time where the processor’s execution units are stalled due to memory accesses missing all caches. It is plotted just to validate the results (not used by the algorithm/implementation).

Almost all benchmarks show improvements over baseline

considering all metrics. The benchmarks with biggest gains are those that are highly memory bounded (low performance scalability). For example, [27] report that external memory is the primary bottleneck for 410.bwaves, 433.milc, 437.leslie3d and 470.lbm. Note how the Ext. Memory Bound correlates with the gains. This makes sense as Turbo isn’t efficient in memory bound phases and it is rather spared for non-memory bound phases.

Benchmarks that are not memory bounded did not have improvements in the metrics as expected as all the programs phases are nearly the same and do not favor running with Turbo in one phase versus the other. As an example of compute bound, 456.hmmmer has loops with tight data-dependent arithmetic instructions [27]. Some of these workloads even showed degradation in the metrics like 444.namd due to instrumentation overhead.

One of the parameters that we tuned manually was the number of instructions threshold on which we skip the handling of a phase, i.e. if at some phase (function or loop) the number of instructions executed is below this threshold

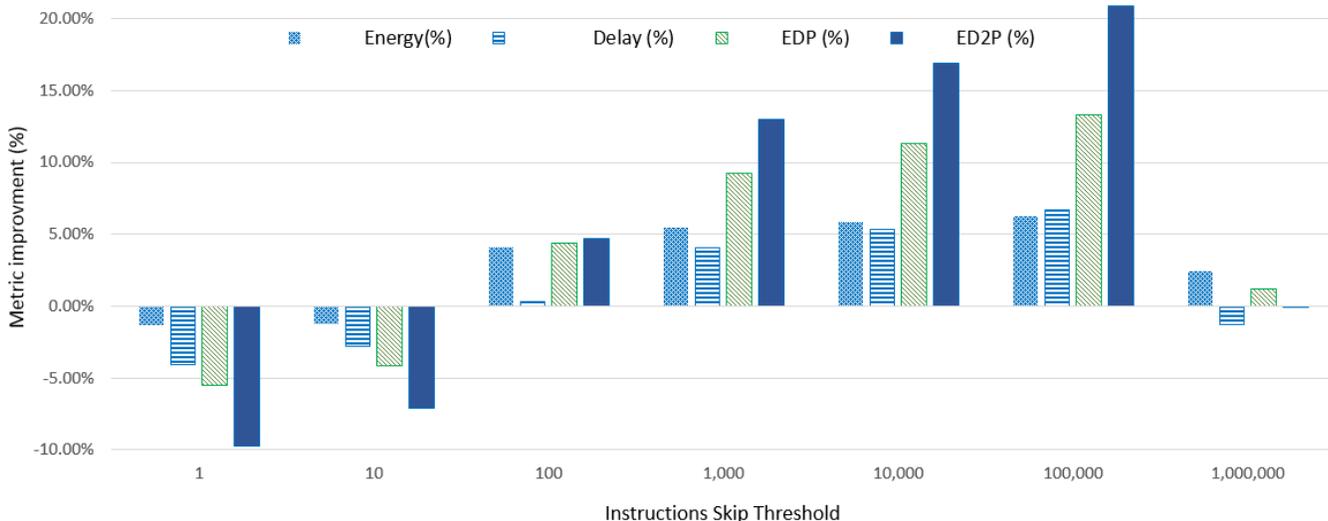


Figure 7: Per-metric gains as a function of skip phase instructions threshold for 429.mcf

then we skip the phase (don't instrument search for it).

Figure 7 shows the performance improvements of the metrics with different instructions skip threshold for one benchmark. The graph shows that low thresholds result in a loss in all metrics; this is expected as such thresholds results in small program-phases being instrumented with the auto-tune code, which would be of high overhead. Best results are achieved near threshold of 100K instructions. It has a good balance between instrumentation overhead and program-phase size leading in a high net metrics gain. At a threshold of 1M we see that some metrics have small improvements while others have loses; this occurs since fewer phases are handled (most phases are skipped) while the initial overhead of measuring the long-phases contributed to the metrics' losses. Note there is high chance for long phases to combine functions/loops with different characteristics.

The Auto-tuning algorithm uses the highest granularity of frequency steps supported by the processor (the difference between P_i and P_{i+1} P-states is 100MHz). This results in high search overhead at the *search phase* in some cases, as the algorithm traverses over the whole range from P_0 to P_n (12 options in our case- from 1900MHz down to 800MHz). Figure 8 shows the ED²P metric while using few frequency-steps (steps of 100MHz, 200MHz, 300MHz, and 400MHz are used). For the less-scalable benchmarks, e.g. 462.libquantum, the gain was reduced with bigger steps; this is mainly due to the fact that a non-optimal frequency was chosen. For middle benchmarks, changing the steps didn't affect the gain much as in case of 436.cactusADM (at 100MHz and 200MHz). For most-scalable benchmarks, e.g. 435.gromacs, the ED²P metric loses with 100MHz step, while we start to see small gains when bigger steps are used. This can be explained as the search overhead was reduced significantly with bigger frequency steps.

The results in Figure 6 are compared to baseline of the balanced performance with energy consumption (setting IA32_ENERGY_PERF_BIAS MSR to value 7) configuration of Intel's energy-efficiency algorithm. Such configuration is relevant to metrics that combine performance (1/delay) with energy like EDP or ED²P.

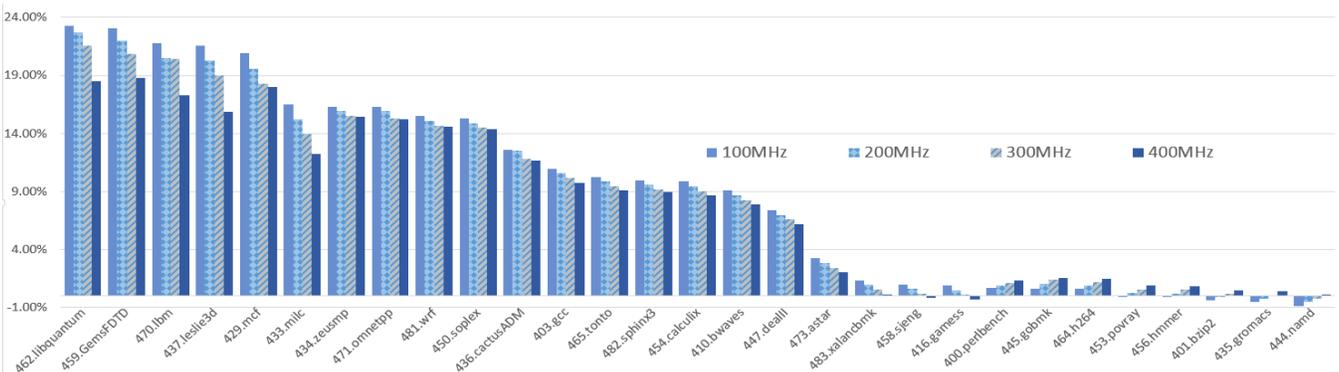


Figure 8: ED²P metric improvement with 100MHz, 200MHz, 300MHz and 400MHz frequency search steps

In Figure 9 we show the various metrics gains with three configurations of Intel's energy-efficiency algorithm for 462.libquantum benchmark. The configurations are Performance, Balanced and Energy-Saver with IA32_ENERGY_PERF_BIAS MSR values of 0, 7 and 15, respectively. Figure 9 shows that at the Performance configuration our framework has the height gain at the energy metric versus the other configurations. While the Energy-Saver configuration our framework has the highest savings at the delay metric. At the Balanced configuration the metrics of ED and ED²P are highest among the three configurations as expected.

To enhance the algorithm and reduce its overhead the following ideas can be explored in a future work:

- Do a binary search to find the frequency work point that will minimize the energy efficiency metric
- Execute a fixed number of search steps, e.g. try just the first 5 frequency options starting from the default one and pick the one that minimizes the energy efficiency metric along the checked options.
- We've demonstrated less-granular tuning in the search phase can reduce overhead (while it might not choose the best performing frequency for the particular benchmark). Further adaptation seems promising, e.g. to choose the step-size dynamically based on current phase characteristics like code size, execution time memory bound-ness and so on.
- Current implementation skips functions with small size after passing first function's execution. This is done in case the function has low instructions count. This process can be optimized by doing it at compile-time, as the code size can be estimated at compile-time in some conditions. This should be done while taking into account calls to other functions from within the original function or loops that exist at the function body. For functions that are marked as short we will not instrument auto-tuning code to them and this will save code size and latency.

RELATED WORK

Although some works [2,3,4] point out that the DVFS gain

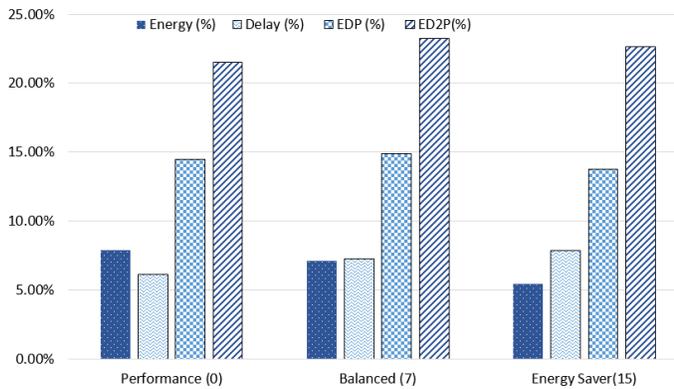


Figure 9: Metrics gains at difference policies of the Intel’s Energy-Efficiency algorithm for 462.libquantum benchmark

is reducing with new process technologies, we do see good energy efficiency gain by applying DVFS. This gain is further increased by Turbo which facilitates high voltage and power that exceed the processor’s TDP. Despite its high power overhead Turbo boosts performance [5].

Previous works have used dynamic methods to improve energy efficiency; Wu et al. [23] developed a dynamic compilation method using Just-In-Time (JIT) compiler that adds code to potential program phases (functions and loop), and activates DVFS as a function of memory bound-ness of each phase. This method has high performance and energy overhead as it activates JIT in runtime to optimize every phase, and requires manual tuning of the memory bound-ness thresholds. In comparison, our method has lower overhead as the compilation is done statically while the Auto-Tuning code figures out the tuning parameters dynamically. Moreover, our method takes into account the platform’s dynamic changes as opposed to looking just at the memory bound-ness which is less accurate with respect to system level optimization.

Koukos et al. [3] proposed a dynamic method for separating memory access phases and execution phases, where low frequency is applied for access phases and high-frequency is applied for execution phases. The technique has high overhead since it requires running each phase twice: first run with only memory accesses and associated address calculations to prefetch the data of that phase to caches; whereas the second run has both memory and execution instructions. The evaluation was done through a simulator, an assumption that DVFS transitions overhead is near zero was made, and the separation of access- and execute-phases was done manually. Jimborean et al. [4] presented a compiler-based method to separate access- and execute-phases. Our method searches for long-enough program phases (skip short ones) to which it applies the Auto-Tuning Algorithm while considering platform changes. Our evaluation uses measurements on production systems and does not assume zero overhead for DVFS transition or Energy counters query. Nonetheless, low overhead would

likely increase the gain and will give better opportunities for tuning short phases.

Sasaki et al. [24] used other hardware performance information available to the operating system to make frequency change decisions. Their DVFS algorithm is based on statistical analysis of performance counters. By predicting the performance, the processor selects the lowest possible frequency that can maintain the performance degradation to a specified ratio. Their technique requires compiler support to insert code for performance prediction, static analysis and per-platform tuning (to build the performance model). In comparison, our method has an auto-tuning algorithm that takes into account dynamic actual platform and workload characteristics with no need for per-platform calibration.

Another approach to dynamically set DVFS performance levels is to use a Performance Monitoring Unit (PMU) to detect when it is possible to achieve sub-linear performance degradation. Isci et al. [25] use phase categories calculated using a metric for Memory Operations per micro-operation. Each phase category is linked to a DVFS policy that attempts to minimize EDP. This approach requires per-platform tuning and does not take into account package level energy.

Rotem et al. [22] presented an algorithm that finds an optimal voltage and frequency operational point of the processor in order to achieve minimum energy for the computing platform. The calibration is (again) per-platform and based on static profiling data, which was also used to validate the algorithm using a fixed power model. Our method has an auto-tuning algorithm, and is more comprehensive where it can optimize any user-supplied metric (not restricted to Energy).

The available Running-Average-Power-Limit (RAPL[17]) energy counters account for all the cores together. There is no option readily accessible that allows reading the counters per-core or per-thread, although we believe that this option will be available in the future. Several studies have been done for CPU performance modeling per-core and per-thread energy using performance counters, Bellsoa’s work [13] shows the linear correlation of hardware events and energy. Singh et al. [14] achieves a run-time per-core power estimation of multithread and multi-program workloads using the top-down method [15]. They categorize the processor’s hardware events into four classes (because their environment platform has only four performance counters). Isci and Martonosi [16] decompose CPU into 22 power breakdowns based on functional units which is a typical bottom-up approach [15]. Following that, they present a per-unit power estimation devised from performance counters.

CONCLUSIONS AND FUTURE WORK

In this work DOEE was developed - a novel method that optimizes processor features for energy efficiency using user-supplied metrics. The optimization is dynamic considering the runtime characteristics of the workload and

the platform. We demonstrate that energy-efficiency optimization is a challenging problem that is hard to solve using hardware-only methods; software hints are essential for accurate optimizations. The evaluation suggests our method outperforms Intel's energy-efficient algorithm implemented by the processors firmware.

We believe that future architectures will exploit software-hardware co-design to raise energy-efficiency of computing systems. We hope to see enhancements at the software-hardware interface, such as DVFS control and the processor's telemetry reading which would enable further enhancements to dynamic optimization algorithms.

Even though the presented auto-tuning (simple) algorithm showed significant savings at the various energy efficiency metrics; further gains seem possible. The algorithm might have high overhead in cases that the function or loop block will be executed a few times, or in case that we have many options for the search. In our current evaluation, the processor frequency range is 800-1900MHz where each frequency bin is 100MHz (12 options in total to search). In other processors the range might be higher and more options will exist which would likely add more overhead to the exploration stage. In addition the current framework does not handle multithreading and multi-core interferences due to contradicting frequency requests of the various threads, for example when the auto-tuning decided to go to high frequency at one core while the opposite was chosen at the other thread, today the CPU will take the maximum request and raise the CPU frequency for both which might not be the most energy efficient at some metric. At our future work we are planning to address these issues and enhance the presented framework.

REFERENCES

- [1] Intel® Corporation. Intel® Turbo Boost Technology in Intel® Core™ Microarchitecture (Nehalem) Based Processors. Whitepaper, Intel® Corporation, November 2008.
- [2] Le Sueur, Etienne, and Gernot Heiser. "Dynamic voltage and frequency scaling: The laws of diminishing returns." Proceedings of the 2010 international conference on Power aware computing and systems. USENIX Association, 2010.
- [3] Koukos, Konstantinos, et al. "Towards more efficient execution: a decoupled access-execute approach." Proceedings of the 27th international ACM conference on International conference on supercomputing. ACM, 2013.
- [4] Jimborean, Alexandra, et al. "Fix the code. Don't tweak the hardware: A new compiler approach to Voltage-Frequency scaling." Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization. ACM, 2014.
- [5] Charles, James, et al. "Evaluation of the Intel® Core™ i7 Turbo Boost feature." Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on. IEEE, 2009.
- [6] D. Dunn, "The best and worst cities for data centers," Information Week, Oct. 23, 2006 edition.
- [7] Grochowski, Ed, and Murali Annavam. "Energy per instruction trends in Intel microprocessors." Technology@ Intel Magazine 4.3 (2006): 1-8.
- [8] R. Gonzalez and M. Horowitz, "Energy dissipation in general purpose microprocessors," IEEE J. Solid-State Circuits, Vol. 31, No. 9, Sept. 1996, pp. 1277-1284.
- [9] Zyuban, Victor, et al. "Integrated analysis of power and performance for pipelined microprocessors." Computers, IEEE Transactions on 53.8 (2004): 1004-1016.
- [10] Brooks, David M., et al. "Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors." Micro, IEEE 20.6 (2000): 26-44.
- [11] Flynn, M., Patrick Hung, and Kevin W. Rudd. "Deep submicron microprocessor design issues." Micro, IEEE 19.4 (1999): 11-22.
- [12] Rotem, Efraim, et al. "Power-management architecture of the intel microarchitecture code-named sandy bridge." IEEE Micro 32.2 (2012): 0020-27.
- [13] Bellosa, F.: The benefits of event: driven energy accounting in power-sensitive systems. In: Proceedings of the 9th Workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System, pp. 37-42. ACM (2000)
- [14] Singh, K., Bhadauria, M., McKee, S.A.: Real time power estimation and thread scheduling via performance counters. ACM SIGARCH Computer Architecture News 37(2), 46-55 (2009).
- [15] Bertran, R., González, M., Martorell, X., et al.: Counter-Based Power Modeling Methods: Top-Down vs. Bottom-Up. The Computer Journal 56(2), 198-213 (2013).
- [16] Isci, C., Martonosi, M.: Runtime power monitoring in high-end processors: Methodology and empirical data. In: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, p. 93. IEEE Computer Society (2003).
- [17] Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3, Section 14.9 (as of November 2014).
- [18] Lattner, Chris, and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation." Code Generation and Optimization, 2004. CGO 2004. International Symposium on. IEEE, 2004.
- [19] James, Dick. "Intel Ivy Bridge unveiled—The first commercial tri-gate, high-k, metal-gate CPU." Custom Integrated Circuits Conference (CICC), 2012 IEEE. IEEE, 2012.
- [20] Standard Performance Evaluation Corporation, [online], Available: www.spec.org/
- [21] Jain, Tarush, and Tanmay Agrawal. "The Haswell Microarchitecture-4th Generation Processor."
- [22] Rotem, Efraim, et al. "Energy Aware Race to Halt: A Down to EARTH Approach for Platform Energy Management." (2012): 1-1.
- [23] Wu, Qiang, et al. "A dynamic compilation framework for controlling microprocessor energy and performance." Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture. IEEE Computer Society, 2005.
- [24] Sasaki, Hiroshi, et al. "An intra-task dvfs technique based on statistical analysis of hardware events." Proceedings of the 4th international conference on computing frontiers. ACM, 2007.
- [25] Isci, Canturk, Gilberto Contreras, and Margaret Martonosi. "Live, runtime phase monitoring and prediction on real systems with application to dynamic power management." Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture. IEEE Computer Society, 2006.
- [26] Jawad Haj-Yihia, LLVM compiler tool for raising energy-efficiency, 2014, from Haifa University: https://drive.google.com/open?id=0B3IgzCqRS5Q_Yi1PbFZCTHpiMEU&authuser=0
- [27] A. Yasin, "A Top-Down Method for Performance Analysis and Counters Architecture," in Performance Analysis of Systems and Software (ISPASS), IEEE International Symposium on, 2014.

Predicting Energy Consumption Relevant Indicators of Strong Scaling HPC Applications for Different Compute Resource Configurations

Hayk Shoukourian^{*,†} Torsten Wilde^{*} Axel Auweter^{*} Arndt Bode^{*,†} Daniele Tafani^{*}

^{*} Leibniz Supercomputing Centre (LRZ) of the Bavarian Academy of Sciences and Humanities
Boltzmannstraße 1, 85748 Garching bei München, Germany

[†] Technische Universität München (TUM), Fakultät für Informatik I10
Boltzmannstraße 3, 85748 Garching bei München, Germany
Hayk.Shoukourian@lrz.de

ABSTRACT

Finding the best energy-performance tradeoffs for High Performance Computing (HPC) applications is a major challenge for many modern supercomputing centers. With the increased focus on data center energy efficiency and the emergence of possible data center power constraints, making the right decision at a given time is becoming more important. A real-world situation like “*can a given 1000 compute node application be executed at a maximum of 2.7 GHz CPU frequency without going over the energy provider defined power band, or the available monthly energy limit?*” is just one example of the types of decisions HPC data centers will face. The previously developed Adaptive Energy and Power Consumption Prediction (AEPCP) model answers this question for the case of a fixed CPU frequency. This paper will extend the AEPCP process to enable the development of analytical models for estimating application execution time, power, and energy consumptions as functions of the number of compute nodes and maximum operating CPU frequency. Based on these analytical models, an adaptive model (*Lightweight Adaptive Consumption Prediction (LACP)*) is presented that implements the extended prediction process. This information allows for improved estimation of potential energy-performance costs and tradeoffs of applications and thus identifies the optimal resource configuration for specific data center boundary conditions.

Author Keywords

consumption modeling and prediction; compute node number and CPU frequency; LACP; power and energy capping; HPC

ACM Classification Keywords

I.6.5 SIMULATION AND MODELING: Model Development

INTRODUCTION

An exponential increase in computing power is making energy efficiency a primary concern for many modern data centers. High energy consumption not only converts into high operating costs, but also transforms into high carbon emissions and serves as a limiting factor for procuring new High Performance Computing (HPC) systems. This issue is expected for Exascale computing systems (systems representing the next thirtyfold increase in computing capabilities beyond currently existing Multi-Petascale systems) where the goal is to have a maximum power consumption of 20 MW [12]. This constraint would mean that data centers should move from the currently used CPU/h accounting models to an energy consumption based charging one in order to actively involve the users in the challenge of saving energy. Take the case where a user has a monthly energy budget which resets by the end of the month. The user would require tools that would allow him to budget in advance which configuration to use to run his applications.

Besides the expected challenges, current utility contracts for data centers include both a connection fee (based on the maximum measured power consumption during the year) and a cost per 1 kWh. Violations of power band boundaries for both the minimum and maximum during a specific window will lead to an unnecessary increase in the connection fee for the year. Consuming less or more energy can lead to additional costs. Therefore, data centers have an increased interest in preventing, for example, power spikes which are increasingly likely due to the higher difference between idle and full power of the HPC systems. For example, SuperMUC, the HPC system at Leibniz Supercomputing Centre (briefly described in subsection “*Compute Platform*”) has an idle power consumption of 700 kW, whereas under Linpack [8] in turbo mode it consumes 3.4 MW.

Energy and power aware resource management and scheduling is one of the keystone requirements on a road to a sustainable, reliable, and environmentally friendly data center. The resource management and scheduling system, being a HPC middleware, is responsible for assignment of compute resources to the queued applications, i.e. applications waiting for execution. Since scientific applications vary, they all have different resource configurations that would lead to an optimal energy-performance ratio. The determination of this

ratio requires the knowledge of the application performance, power, and energy profiles under, potentially, all possible configurations. In this paper, the tuple (a, b) , where a represents the number of compute resources and b indicates the maximum CPU frequency of the compute resources, is referred to as *configuration*. Current resource management and scheduling systems have no knowledge of execution time and energy or power consumption of applications to be scheduled for a given configuration.

In order to implement energy-aware resource management and allocation, the information on at least two of the following metrics is required (these two metrics can then be used to derive the third):

- **Time-to-Solution (TtS)**
Indicating the execution time of a given application
- **Energy-to-Solution (EtS)**
Indicating the aggregated energy consumption of a given application
- **Average Power Consumption (APC)**
Indicating the mean power draw of a given application, i.e.
 $APC = \frac{EtS}{TtS}$

These metrics could be different for the same application when executed with different configurations. Figure 1 shows the EtS behavior of a scientific application Hydro¹ for different numbers of compute nodes and CPU frequency configurations when executed on the SuperMUC supercomputer. As can be seen, the EtS notably varies with different resource configurations.

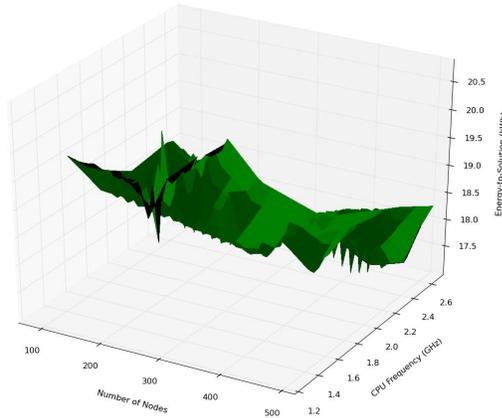


Figure 1: Energy consumption profile of Hydro with different numbers of utilized compute nodes and CPU frequencies

This paper proposes an adaptive model, referred to as *Lightweight Adaptive Consumption Predictor (LACP)*, for predicting values of TtS, EtS, and APC. The suggested LACP model describes the behavior of TtS, EtS, and APC with respect to the number of utilized compute nodes and the global maximum CPU frequency across all the cores of the compute nodes, taking as an input the history execution

¹Briefly described in subsection “Applications”.

time/power/energy profiles of the application under the assumption of a fixed application input problem size (i.e. for applications demonstrating *strong scaling*). This model is application independent but its execution provides application specific results. With each additional execution of a specific application the prediction accuracy for this application is improved.

The rest of this paper is organized as follows. Section “*BACKGROUND AND RELATED WORK*” discusses the background information on TtS, EtS, and APC metrics for applications demonstrating strong scaling and presents the related studies. Section “*ADVANCING THE STATE OF THE ART*” analytically approximates the behaviors of TtS, EtS, and APC as functions of the number of compute nodes and the CPU frequency at which all the cores of all compute nodes operate. Section “*THE APPROACH*” gives an overview of the IBM LoadLeveler prediction model, the AEPCP [19] model, and presents the LACP process and model. Section “*LACP VALIDATION*” introduces the compute platform and the two real-world applications, which were used to validate the suggested LACP model, discusses the prediction results, and outlines the use cases of the LACP model in real-world scenarios. And finally, section “*CONCLUSION AND FUTURE WORK*” draws the summary and concludes the paper.

BACKGROUND AND RELATED WORK

Strong scaling was first described analytically by Gene Amdahl in 1967 [2]. According to Amdahl’s law, the total $TtS(n)$ processing time of sequential and parallel parts using n compute resources, can be derived as:

$$TtS(n) = TtS(1) \cdot \left[(1 - p) + \frac{p}{n} \right] \quad (1)$$

where p is the parallel fraction of the application, and $1 - p$ is the serial fraction ².

A study by Woo and Lee [24], considering Amdahl’s law, proposes an analytical model for calculating the average power consumption $APC(n)$ of a given application when executed on n compute resources:

$$APC(n) = \frac{1 + (n - 1) \cdot k \cdot (1 - p)}{(1 - p) + \frac{p}{n}} \quad (2)$$

where k is the fraction of power that is consumed by the compute resource in the idle state ($0 \leq k \leq 1$).

Based on these laws, an adaptive model, referred to as *Application Energy and Power Consumption Prediction (AEPCP)* [19], was developed to predict the execution time, average power, and energy consumptions of a given parallel application for a given number of compute nodes. As can be seen, these laws (and thus the AEPCP model) do not take into account the possibility of CPU frequency modification.

²The application is normalized as 1 and divided into two portions. While the execution time of the parallel portion can be reduced using number of compute nodes, that is not the case for the serial portion.

Ge et al. [10] suggest analytical models to approximate performance and energy cost for a given HPC application on multi-core based power aware systems. The authors in [10] further split the serial portion of the application into processor frequency dependent portion: $(1-p)\alpha_s$, i.e. portion that benefits from faster processor speed; and processor frequency independent portion: $(1-p)(1-\alpha_s)$, i.e. portion that does not benefit from the processor speed³. Similarly, the application parallel portion is split into processor frequency dependent portion: $p\alpha_p$; and processor frequency independent portion: $p(1-\alpha_p)$.

Based on these notations, Ge et al. estimate the parallel execution time for a given configuration of n total number of cores, c per compute node allocated number of cores, and f processor frequency:

$$TtS(n, c, f) = (1-p)(1-\alpha_s + \alpha_s \frac{f_0}{f}) + \frac{p}{n}(1-\alpha_p + \alpha_p \frac{f_0}{f}) + O_{n,c,f} \quad (3)$$

where f_0 is the base CPU frequency and $O_{n,c,f}$ is the parallelization overhead. Since for most supercomputers the compute node is the smallest compute resource a job can use, it is assumed that the number of cores c per compute node is constant and, therefore, the term c is omitted from further discussions and n is further considered as the number of compute nodes.

Although Ge et al. [10] derive the analytical models for power and energy, as well as investigate the energy-performance efficiency of parallel applications, the derivation process of the parameters presented in their models is not automated [10] and in general should be derived through linear regression using the data obtained from experimental benchmarks. It is worth noting that, for example, in order to derive the parallelizable and not parallelizable, or frequency dependent and independent portions of the application, one would require extra tools to detect these application fractions. This could be impractical in real-world scenarios where several hundred applications with different characteristics are used⁴.

A more detailed survey of related work, including available tools, can be found in [19].

ADVANCING THE STATE OF THE ART

This section will analytically refine the TtS, APC, and EtS metrics of an application as functions of the number of compute nodes and the CPU frequency at which all the cores of all compute nodes operate. These refined functions will approximate the TtS, APC, and EtS dependency behaviors of a given application explicitly from the number of compute nodes and maximum operating CPU frequency and will treat the parallelizable and not parallelizable (as well as frequency dependent and independent) portions of the application as constant

³Some of the terminology defined in [10] has been borrowed for consistency reasons.

⁴Which is typically the case for modern HPC data centers.

fitting parameters since, as was mentioned above, the derivation of these fractions for every application could be impractical, if not impossible, in real-world scenarios.

Equation 3 can be further approximated as: $TtS(n, f) \in O(1) + O(\frac{1}{f}) + O(\frac{1}{n}) + O(\frac{1}{nf}) + O(O_{n,f})$ ⁵. The term $O_{n,f}$ can be broken down and approximated in the following way. First, the overhead $O_{n,f}$ arises due to the parallelizable fraction of the application. Second, during the execution of the parallelizable and frequency independent fraction of the application the resulting parallel overhead of n compute resources (nodes) is of the order of $O(n)$. Whereas during the execution of the parallelizable and frequency dependent fraction the parallel overhead is of the order of $O(\frac{n}{f})$, since the overhead time gets reduced as the processor frequency increases. Thus complete parallel slowdown $O_{n,f}$ is of $O(n + \frac{n}{f})$ order, which further means that $TtS(n, f) \in O(\frac{1}{f} + \frac{1}{n} + \frac{1}{nf} + n + \frac{n}{f})$. Therefore, the $TtS(n, f)$ for given n compute nodes and CPU frequency f can be approximated via the following equation:

$$TtS(n, f) = \frac{t_1}{f} + \frac{t_2}{n} + \frac{t_3}{nf} + t_4 n + t_5 \frac{n}{f} + t_6 \quad (4)$$

where all t_i ($1 \leq i \leq 6$) are constant fitting parameters.

Although Ge et al. [10] propose an analytical model for estimating the node power consumption, it requires the additional estimations of dynamic to idle power scaling factors (which could be application dependent) and does not explicitly show the dependency from operating CPU frequency of the compute node.

Since CPU and memory are the major power consuming resources in a typical compute node [20], and also the ones, who could potentially vary with processor frequency, they are treated individually as also done in [10]. The break down of one compute node average power consumption $APC(1, f)$ for a given frequency f can be done in the following way:

$$APC(1, f) = P_{CPU,dynamic} + P_{CPU,idle} + P_{memory,dynamic} + P_{memory,idle} + P_{other} \quad (5)$$

where $P_{CPU,dynamic}$ indicates the power sum of all cores of the compute node when all cores execute some workload; $P_{CPU,idle}$ quantifies the power sum of all cores of the compute node when there is no application running on any of the cores; $P_{memory,dynamic}$ represents the power consumption of compute node dynamic memory; $P_{memory,idle}$ represents the power consumption of compute node memory when it is idling; and P_{other} shows the aggregated power consumption of the other components of the compute node such as hard drives, PCI slots, etc.

In this paper, it is assumed that the operating CPU frequency is changed through the CPU multiplier and not

⁵ $O(g(n)) = \{f(n) | \exists c > 0 \text{ constant and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \forall n \geq n_0\}$ [7]

through the processor base clock. Thus the CPU frequency modification should have no impact on the power consumption of memory, as well as on the power consumption of other compute node components. Therefore, only $P_{CPU,dynamic}$ and $P_{CPU,idle}$ are considered as frequency dependent and all other terms can be treated as constant, i.e. $P_{memory,dynamic} = P_{memory,idle} = P_{other} = O(1)$.

The power consumption of an operating CPU at a given frequency f can be summarized through the following equation (as shown in [23]):

$$\begin{aligned} P_{cpu,dynamic} &= P_{switching} + P_{short-circuit} + P_{static} \\ &= \alpha C_L V_{dd}^2 f + P_{short-circuit} + P_{static} \end{aligned} \quad (6)$$

where α is the switching activity factor, i.e. the probability of the circuit node transition from 0 to 1; C_L is the load capacitance; V_{dd} is the supply voltage; and $P_{short-circuit}$ is the power leakage aroused due to the short-circuit current flowing from supply to ground when both pMOS and nMOS stacks are conducted. A survey by H. Veendrick [22] provides an in-depth discussion of $P_{short-circuit}$ power dissipation. The P_{static} is the static power dissipation of the processor aroused due to the subthreshold leakage, the gate leakage, the junction leakage, and the contention current (as described in [23]).

When further considering the fact that the supply voltage V_{dd} linearly depends on the processor frequency f [16], the following is obtained: $P_{switching} \in O(f^3)$. Since the dynamic power is usually dominated by $P_{switching}$ power of charging and discharging load capacitances as gates switch [23, 5], the $P_{short-circuit}$ and P_{static} can be considered as constant in this case, i.e. $P_{short-circuit} \in O(1)$, $P_{static} \in O(1)$ and thus $P_{cpu,dynamic} \in O(f^3 + 1)$, leading to:

$$P_{cpu,dynamic} = a_1 f^3 + a_2 \quad (7)$$

where a_1 , and a_2 are constant fitting parameters. On the other hand, when a processor is idle, the power dissipation is determined by the leakage, i.e. P_{static} [23]. Due to the dynamic voltage frequency scaling done by the Power Control Unit (PCU) of energy-saving features enabled processors [17], the power supply voltage of the processor is always reduced during the idling periods (thus also the frequency) to an optimal point, and hence, it can be assumed that $P_{static} \in O(1)$. Therefore by using Equation 5 the average power consumption of one node can be derived as: $APC(1, f) \in O(f^3 + 1)$ for a given CPU frequency f , and the average power consumption of n nodes, i.e. $APC(n, f)$, for a given global frequency f will be of order $O(nf^3 + n)$, which further means that:

$$APC(n, f) = k_1 n f^3 + k_2 n + k_3 \quad (8)$$

where k_1 , k_2 , and k_3 are constant fitting parameters. Using the portion split of the application defined above, the following is derived:

- **portion 1 - serial and frequency dependent.** For this case $T_1(n, f) \in O(\frac{1}{f})^6$. Since only 1 node is active during the serial portion execution and the rest $n - 1$ nodes are idling and thus consuming $O(1)$ power, $APC_1(n, f) \in O(f^3 + (n - 1)) = O(f^3 + n)$, and thus $EtS_1(n, f) = TtS_1(n, f)APC_1(n, f) \in O(f^2 + \frac{n}{f})$;
- **portion 2 - serial and frequency independent.** For this case $TtS_2(n, f) \in O(1)^6$. In this case, $APC_2(n, f) = APC_1(n, f) \in O(f^3 + n)$, thus $EtS_2(n, f) \in O(f^3 + n)$;
- **portion 3 - parallel and frequency dependent.** For this case $TtS_3(n, f) \in O(\frac{1}{nf})^6$. In this case, all n compute nodes are active and consuming $O(nf^3 + n)$ power in total. Thus the energy consumption for a given number n of compute nodes and CPU frequency f , is given: $EtS_3(n, f) = O(f^2 + \frac{1}{f})$;
- **portion 4 - parallel and frequency independent.** For this case $TtS_4(n, f) \in O(\frac{1}{n})^6$, and $APC_4(n, f) = APC_3(n, f) = O(nf^3 + n)$. Thus the $EtS_4(n, f)$ energy consumption for this section will be of order $O(f^3 + 1)$ for a given n number of compute nodes and f global CPU frequency;
- **portion 5 - parallel overhead.** For this case $TtS_5(n, f) \in O(\frac{n}{f} + n)^7$. Since all n compute nodes could be active during this period, the total power $APC_5(n, f)$ and energy $EtS_5(n, f)$ consumptions for the parallel overhead portion for given n compute nodes when all are running at a given frequency f is: $APC_5(n, f) = APC_4(n, f) = APC_3(n, f) \in O(nf^3 + n)$; $EtS_5(n, f) = O(n^2 f^2 + \frac{n^2}{f} + n^2 f^3 + n^2)$.

Thus the total EtS energy consumption of an application utilizing n compute nodes, when all of which are running at a given f CPU frequency, can be approximated through the following equation:

$$\begin{aligned} EtS(n, f) &= \sum_i^5 EtS_i(n, f) = b_1 f^2 + b_2 \frac{n}{f} + b_3 f^3 + b_4 n + \\ &\quad \frac{b_5}{f} + b_6 + b_7 n^2 f^2 + b_8 \frac{n^2}{f} + b_9 n^2 f^3 + b_{10} n^2 \end{aligned} \quad (9)$$

where all b_i ($1 \leq i \leq 10$) are constant fitting parameters. Note, that when an application demonstrates ideal scaling, the parallelizable portion p equals to 1 and thus the serial portion (i.e. $1 - p$) of the application becomes 0. Assuming also that the parallel slowdown is negligible during the ideal scaling, the Equation 9 can be rewritten in the following way:

⁶Implied from Equation 3.

⁷As discussed for Equation 4.

$$\begin{aligned}
EtS(n, f) &= EtS_3(n, f) + EtS_4(n, f) \\
&= c_1 f^3 + c_2 f^2 + \frac{c_3}{f} + c_4
\end{aligned} \tag{10}$$

where all c_i ($1 \leq i \leq 4$) are constants. This further means that, in the case of ideal scaling, the usage of more compute nodes would potentially increase the performance⁸ without any additional energy costs, as also mentioned in [10]. While the increase in CPU frequency would potentially bring a cubic rise in the energy costs⁹.

As can be seen, in the case of a **fixed** n number of compute nodes, the $TtS(n, f) \in O(\frac{1}{f})$ (Equation 4) and $APC(n, f) \in O(f^3)$ (Equation 8). This further means that

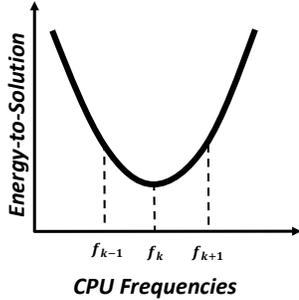


Figure 2: The EtS behavior from the operating CPU frequency in case of a fixed n number of compute nodes

Since $f_{k-1} < f_k$ then $APC(n, f_{k-1}) < APC(n, f_k)$, while, as can be seen from the Figure 2, $EtS(n, f_{k-1}) > EtS(n, f_k)$, which further means that *minimizing the energy consumption does not necessarily lead to a corresponding power minimization*. On the other hand, since $f_k < f_{k+1}$ then $TtS(n, f_k) < TtS(n, f_{k+1})$, while, as can be seen from the Figure 2, $EtS(n, f_k) < EtS(n, f_{k+1})$, which further means that *minimizing the energy consumption does not necessarily lead to an execution time benefit*.

THE APPROACH

IBM LoadLeveler and AEPCP alone

LoadLeveler [1] is a resource and job management system developed by IBM. LoadLeveler has a prediction model [6] for application energy, power, and runtime estimation when executed using different CPU frequencies for a fixed number of compute nodes. It was shown that the predictions accomplished through the LoadLeveler model have a high accuracy rate [4].

⁸In other words, potentially decrease the runtime, since in the case of ideal scaling: $TtS(n, f) \in O(\frac{1}{nf} + \frac{1}{n})$.

⁹Under the assumption that the supply voltage varies linearly with the operating CPU frequency.

Figure 3 shows a sample output from the LoadLeveler prediction interface for the Hydro application-benchmark (described in subsection “Applications”), when executed on 210 compute nodes.

Nominal Frequency: 2.70 GHz Default Frequency: 2.30 GHz Node's DC Energy Use: 0.081939 kWh Execution Time: 1868 Seconds						
Frequency(GHz)	EstDCEngCons(kWh)	DCEngVar(%)	EstTime(Sec)	TimeVar(%)	Power(W)	
2.70	0.086225	5.23	1606	-14.03	193.28	
2.60	0.084873	3.58	1663	-10.97	183.73	
2.50	0.083688	2.13	1727	-7.55	174.45	
2.40	0.082608	0.82	1792	-4.07	165.95	
2.30	0.081939	0.00	1868	0.00	157.91	
2.20	0.081078	-1.05	1946	4.18	149.99	
2.10	0.080695	-1.52	2035	8.94	142.75	
2.00	0.080486	-1.77	2132	14.13	135.91	
1.90	0.080723	-1.48	2248	20.34	129.27	
1.80	0.081092	-1.03	2370	26.87	123.18	
1.70	0.081685	-0.31	2506	34.15	117.35	
1.60	0.082534	0.73	2659	42.34	111.74	
1.50	0.083713	2.17	2833	51.66	106.38	
1.40	0.085082	3.84	3026	61.99	101.22	
1.30	0.087011	6.19	3249	73.93	96.41	
1.20	0.089608	9.36	3511	87.96	91.88	

Figure 3: Sample output of LoadLeveler CPU prediction results for Hydro when executed on 210 compute nodes

As can be seen, for a specific application execution with a fixed number of compute nodes, LoadLeveler provides the per node energy (second column, i.e. “EstDCEngCons(kWh)”) and power consumption (the last, sixth column, i.e. “Power (W)”) as well as complete runtime (fourth column, i.e. “EstTime(Sec)”) estimations for the application when executed with different CPU frequencies. One could argue that just by multiplying these results with different numbers of compute nodes, the information regarding TtS/EtS/APC metrics for that numbers of compute nodes can be estimated. Table 1 shows that this is not the case.

The *first* column of Table 1 shows the compute node number used for comparison; the *second* column shows the CPU frequencies; the *third* column shows the measured energy consumption for a given configuration; the *fourth* column shows the predicted energy consumption using the above mentioned multiplication method for the corresponding configuration; and finally, the *fifth* column shows the prediction error rate of the used method calculated as $\frac{\text{measured} - \text{predicted}}{\text{measured}} * 100$.

Number of Compute Nodes	CPU Frequency (GHz)	Measured EtS Value (kWh)	LoadLeveler Predicted EtS Value (kWh)	Prediction Error Rate (%)
150	1.6	18.16	12.38	31.8
	1.8	17.75	12.16	31.49
	2.3	17.76	12.29	30.79
	2.7	18.66	12.93	30.7
210	1.6	17.65	17.33	1.8
	1.8	17.21	17.03	1.05
	2.3	17.21	17.21	0
	2.7	17.97	18.11	0.78
320	1.6	18.48	26.41	43
	1.8	18.31	25.95	41.7
	2.3	17	26.22	54.23
	2.7	17.57	27.6	57

Table 1: Prediction results for Hydro using data obtained from IBM LoadLeveler for 210 compute nodes

The “-” highlighted circle in Table 1 illustrates the LoadLeveler available data for 210 compute nodes. The rows above and below the “-” highlighted circle in Table 1 show the predicted EtS for different numbers of compute nodes (150 and 320)¹⁰ and different CPU frequencies. As

¹⁰These numbers were chosen on a random basis.

can be seen, the prediction error for the simple multiplication method, when using the LoadLeveler predicted data, is more than 30% for less/greater than 210 count of compute nodes.

Adaptive Energy and Power Consumption Prediction (AEPCP) [19] is another model capable of predicting the power and energy consumption of parallel HPC applications for different numbers of compute nodes under the assumption of fixed maximum allowed CPU frequency. It is application independent and describes the behavior of power and energy with respect to the number of utilized compute nodes, taking as an input the available history power/energy data of an application. It *provides a generic solution* that can be used for each application *but it produces an application specific result*. The AEPCP model allows for ahead of time power and energy consumption prediction and adapts with each additional execution of the application improving the associated prediction accuracy. However, the AEPCP model does not support predictions with respect to different CPU frequencies.

The *Lightweight Adaptive Consumption Predictor* (LACP), discussed in the next subsection, is the extension of the AEPCP model allowing the prediction of the energy and power consumption as well as the runtime of a given application for an arbitrary (number of compute nodes, maximum CPU frequency) configuration.

Lightweight Adaptive Consumption Predictor (LACP)

LACP Process: Figure 4 shows the prediction process [11] of the Lightweight Adaptive Consumption Predictor (LACP). As was the case of the AEPCP model [19], the LACP process takes as input: (i) the application identifier (used to uniquely identify the application); and (ii) the number of compute resources (e.g. CPU, compute nodes, accelerators, etc.) planned for application utilization. In addition to the inputs required by the AEPCP process, the LACP process also requires the maximum CPU frequency at which all the cores of the compute resources will operate.

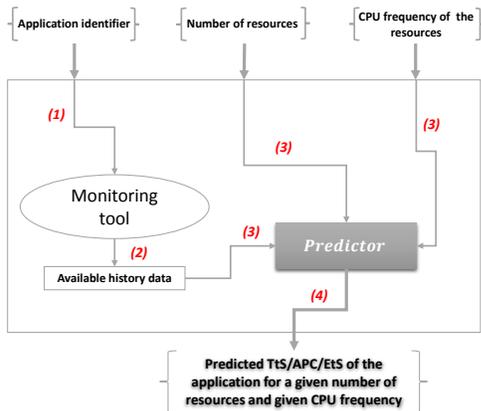


Figure 4: Overview of the LACP process

The LACP process (Figure 4) uses the application identifier to uniquely query the application execution relevant history information from the system monitoring tool (step (1)). Then, this history information (step (2)), together with the number

of compute nodes and the given CPU frequency, is passed to the predictor (step (3)) for corresponding TtS/APC/EtS prediction. Based on this input data, the predictor then estimates the TtS/APC/EtS consumption values (step (4)).

LACP Model: Figure 5 describes the LACP model based on the prediction process (Figure 4). The LACP model requires three inputs, namely:

- **application energy tag prefix:** used as an application unique identifier. The application energy tag is a unique identifier for the application, supported by the LoadLeveler, and is specified by the user on a unique-per-application basis. A new energy tag needs to be specified by the user each time an application is submitted for execution with a different compute node configuration. This way LoadLeveler (the resource management system used on SuperMUC) generates the corresponding prediction data for that specific node number configuration. This new tag should have a unique prefix to identify all the existing executions of that application. For example, *myAutoCrashSimulationETag_178nodes* and *myAutoCrashSimulationETag_317nodes* could be two energy tags for an application having a unique *myAutoCrashSimulationETag* prefix identifying them both;
- **number of compute nodes:** used as the number of compute resources. A compute node is the smallest compute unit available to an application on the SuperMUC super-computer (briefly described in subsection “*Compute Platform*”) which was used to validate the LACP model;
- **maximum CPU frequency:** enforced for all used compute nodes.

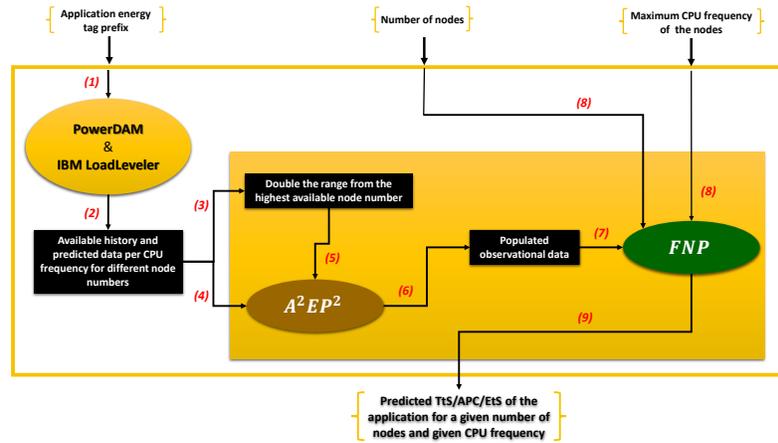


Figure 5: Overview of the LACP model

The application energy tag prefix is used by the system monitoring tool to retrieve the history energy/power/runtime relevant profile data of the application executions for different configurations (step (1), Figure 5). In this case, Power Data Aggregation Monitor (PowerDAM) [18] is used to extract this data. PowerDAM is a unified energy measurement and evaluation toolset aimed towards collecting and correlating

energy consumption-relevant data from different aspects of the HPC data center (e.g. environmental, site infrastructure, IT systems, running applications, etc.). Next, the available LoadLeveler prediction data is compiled with the history data (step (2)).

Once this compiled data is obtained, the maximum and minimum number of compute nodes are determined within this data, correspondingly doubled and halved (step (3)), and together with the compiled history data, passed to the Adaptive Application Energy and Power Predictor (A^2EP^2) (step (4) and step (5)). A^2EP^2 is a predictor for estimating the application EtS/APC consumption for any given number of compute nodes and is part of AEPCP model [19]. The A^2EP^2 was extended to allow for the TtS metric prediction, using the fact that, in the case of fixed CPU frequency, the execution time $T(n)$ for n number of compute nodes, for strong scaling, can be approximated as $T(n) = \frac{a_1}{n} + a_2$, where a_1 and a_2 are constant fitting parameters¹¹. This is achieved by using an interpolation technique to estimate the a_1 and a_2 constants in $T(n)$ with respect to different compute node numbers. A detailed description on A^2EP^2 can be found in [19].

Using the data from steps (4) and (5), the A^2EP^2 extends this data set in the following way. For a fixed frequency f , A^2EP^2 is used to predict the TtS, APC, and EtS for all unknown number of compute nodes¹² which are in the range of $[\frac{\min \text{ observed node number}}{2}; 2 \times \max \text{ observed node number}]$. The double/half distance for prediction is used, since the A^2EP^2 has been proven to provide acceptable results within that range [19]. The A^2EP^2 is then used for all the LoadLeveler supported frequency values.

This extended data (step (6)) is then passed to the *Frequency and Node Number Predictor* (FNP) (step (7)). Then the FNP, using the derived analytical behaviors of TtS (Equation 4), APC (Equation 8), and EtS (Equation 9) in section “*ADVANCING THE STATE OF THE ART*”, estimates the exact values of the coefficients using polynomial multivariate interpolation. Once these coefficients are determined, the FNP calculates the TtS/APC/EtS values of the application for a given number of compute nodes and given CPU frequency (steps (8) and (9); Figure 5).

LACP VALIDATION

The LACP model was validated using two real-world application benchmarks executed on the SuperMUC [14] supercomputer. The following subsections describe the SuperMUC supercomputer and the used application benchmarks, discuss the LACP prediction results, as well as show the applicability of the LACP model in different emerging real-world use case scenarios.

Compute Platform

SuperMUC, operated by Leibniz Supercomputing Centre (LRZ) [14], is the 12th fastest supercomputer in the world according to the Top500 [21] June 2014 rankings. It is an

¹¹Implies from Amdahl’s Law (Equation 1).

¹²In other words, number of compute nodes which are not in the input data set.

iDataPlex *DX360M4*, Xeon *E5 – 2680 8C* system with 155.656 processor cores in 9421 compute nodes connected via *FDR10 InfiniBand* [14]. It uses IBM LoadLeveler [1] as a resource and job management system. SuperMUC’s active components (e.g. processors, memory, etc.) are directly cooled with an inlet water temperature of up to 40° Celsius. SuperMUC is a GCS (Gauss Center for Supercomputing) [9] infrastructure system and one of the PRACE (Partnership for Advanced Computing in Europe) [15] Tier0 systems.

The validation was performed on a thin node island of SuperMUC consisting of 512 nodes. Each thin node has 2×8 core Sandy Bridge-EP Intel Xeon *E5 – 2680 8C* processors having a maximum allowed operating frequency of 2.7 GHz. The measurements obtained from SuperMUC’s paddle cards, as presented in [19] and in [4], show a high accuracy thus dismissing the need for a re-execution of any benchmark for any given configuration.

Applications

Hydro: Hydro [13] is a computational fluid dynamics 2D code for solving the compressible Euler equations of hydrodynamics.

EPOCH: EPOCH [3] is a plasma physics code based upon the particle push and field update algorithms. It uses the MPI-parallelized explicit 2nd order relativistic particle-in-cell method, including a dynamic MPI load balancing option.

EtS - Tackling the Energy Capping

Assume a user has a monthly energy budget of 71 kWh. He has executed Hydro using: 105 compute nodes; 170 compute nodes; 240 compute nodes, and 370 compute nodes¹³, always using the maximum CPU frequency of 2.3 GHz. All these runs in total consume 52.24 kWh leaving him with around 18 kWh in his energy budget. Now he wants to run Hydro on 450 compute nodes. The question he needs to answer is: *can I run Hydro on 450 compute nodes without going over my available energy budget?* Currently none of the available resource management and scheduling systems can answer this question. By using the LACP model it is possible to find an answer.

Table 2 illustrates the PowerDAM tracked TtS/APC/EtS values of Hydro when executed by the user on 105, 170, 240, and 370 compute nodes, all at CPU frequency of 2.3 GHz.

Number of Compute Nodes	Maximum CPU Frequency (GHz)	Measured EtS Value (kWh)	Measured APC Value (Watt)	Measured TtS Value (min)
105	2.3	19.28	17736.55	65.22
170	2.3	17.78	27375	38.98
240	2.3	17.36	37659	27.66
370	2.3	17.1	55817	18.38

Table 2: PowerDAM tracked Hydro history data

Figure 6 shows the LACP EtS prediction results for 450 compute nodes for different maximum CPU frequencies. The first bar (when counted from left to right), colored in gray, shows the measured EtS, and the second bar (colored in red) shows

¹³All these numbers of compute nodes were randomly chosen.

the LACP predicted EtS¹⁴. As can be seen, the highest prediction error is then 7% with respect to different CPU frequen-

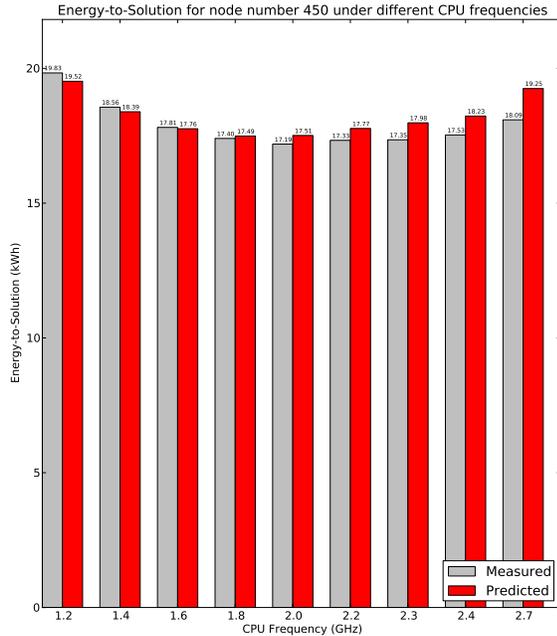


Figure 6: LACP EtS prediction results for 450 compute nodes Hydro execution under different CPU frequencies
Note: available data points are for 105, 170, 240, and 370 compute nodes

Using the LACP prediction results (Figure 6) the posed user question can be answered. The execution of Hydro using 450 compute nodes is possible within the 18 kWh energy budget which is left, if the user restricts the maximum CPU frequency to the [1.6; 2.3] range, since any execution that uses (450, f) configuration, where $f \in [1.6; 2.3]$, will have predicted EtS value of less than 18 kWh (red bar, Figure 6) and thus will not violate the user’s budget.

APC - Tackling the Power Capping

The introduction discussed the need for data centers to avoid power spikes. Assume the data center operator has set a power limit of 65 kW that must not be violated at any given point in time. Continuing our example with Hydro from subsection “EtS - Tackling the Energy Capping”, the question to answer is: *is it possible to satisfy the user request and run Hydro using 450 compute nodes without power boundary violation?* Currently, none of the available resource management and scheduling systems can answer this question. The previously suggested AEPCCP model [19] can answer that question, but only for the fixed CPU frequency, in this case for 2.3 GHz, since all the user executions of Hydro were conducted at a maximum frequency of 2.3 GHz (Table 2). The LACP model provides a broader answer.

Figure 7 shows the LACP APC prediction results running Hydro on 450 compute nodes for different frequencies. The first

¹⁴Subsequently presented EtS bar charts have the same structure.

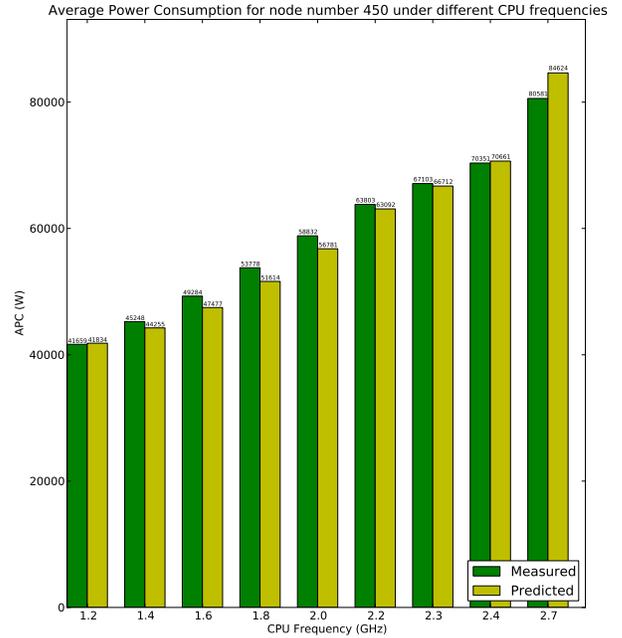


Figure 7: LACP APC prediction results for 450 compute nodes Hydro execution under different CPU frequencies
Note: available data points are for 105, 170, 240, and 370 compute nodes

bar (colored in green) shows the measured APC, and the second bar (colored in yellow) shows the predicted APC¹⁵. Several things can be observed from this bar chart. First, the execution of Hydro utilizing 450 compute nodes would not be allowed when using the AEPCCP model APC prediction. As can be seen, the (450, 2.3) configuration has a predicted APC of 66,712 Watt and thus violates the predefined 65 kW power constraint. Since LACP is also capable of predicting the APC values for different CPU frequencies (Figure 7), its usage shows that the execution of Hydro utilizing 450 compute nodes is possible, as long as the maximum CPU frequency is selected from the [1.2; 2.2] GHz range, since for any frequency $f \in [1.2; 2.2]$ the predicted APC(450, f) is less than 65 kW (yellow bar).

TtS - Tackling Execution Time vs Energy Consumption Tradeoff

As seen in subsection “EtS - Tackling the Energy Capping” and in subsection “APC - Tackling the Power Capping”, several configurations can satisfy given energy or power consumption constraints. It is obvious that in most cases the user would prefer to execute their application at the maximum possible CPU frequency, since this would lead to execution time reduction. On the other hand, the energy consumed by an application changes with the CPU frequency (Figure 6). This makes the knowledge on application execution time with respect to various potentially unknown (i.e. not executed before) configurations important for determining the possible tradeoffs. From Figure 6 it can be seen that the user could run Hydro using, for example, (450, 2.3) and (450, 1.8) configurations and, as was discussed in subsection “EtS - Tackling

¹⁵Subsequently presented APC bar charts have the same structure.

the Energy Capping”, the execution with either configuration would not run over the 18 kWh energy budget. The execution of Hydro using (450, 1.8) configuration will cost less energy than the execution with (450, 2.3) configuration. So, which configuration should the user select to run Hydro?

Again, currently available scheduling and management systems cannot support the user in making this decision.

Figure 8 shows the LACP TtS prediction results when executed at different CPU frequencies. The first bar, colored in gray and labeled as *Transitively Predicted (EtS/APC)*, illustrates the prediction results obtained using the data presented in Figure 6 - Figure 7 (since $TtS = \frac{EtS}{APC}$); the second bar (colored in blue) shows the measured TtS; and finally the third bar (colored in white) labeled as *Directly Predicted* shows the case when LACP uses the available TtS data from Table 2 for the configuration.

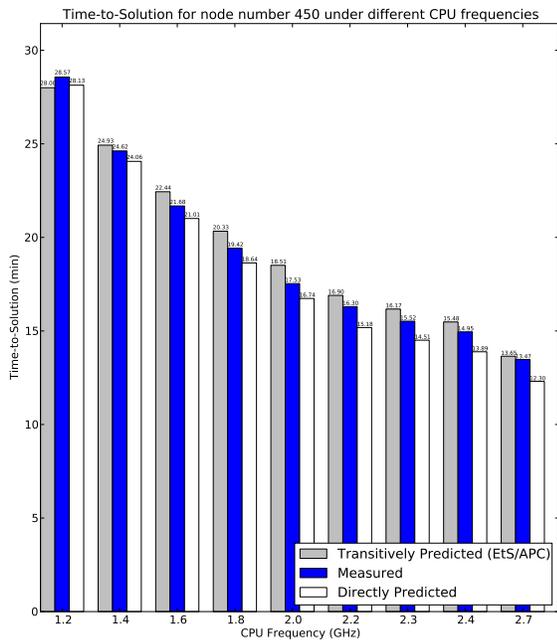


Figure 8: LACP TtS prediction results for 450 compute nodes Hydro execution under different CPU frequencies

Note: available data points are for 105, 170, 240, and 370 compute nodes

As can be seen, in the case of the execution with (450, 1.8) configuration the execution time will be 18.64 minutes (directly predicted, white chart), whereas in the case of execution using the (450, 2.3) configuration the predicted execution time is 14.51 minutes. Combining this information with the EtS prediction results (Figure 6) the user can decide to trade a 4 minute longer runtime for a 0.49 kWh in energy savings.

The information on the possible execution time for a given configuration can yield tighter bounds for the wall-clock time the user specifies for the HPC scheduling system. This in turn could improve the efficiency of the backfilling process of the

¹⁶Subsequently presented TtS bar charts have the same structure.

resource management and scheduling systems, thus minimizing the waiting time¹⁷ for the users.

Tackling A Combined Real World Scenario

This subsection discusses the scheduling possibility using EPOCH considering the possible tradeoffs between user energy budget and execution time under a predefined system power cap. Assume the following scenario. The user executed EPOCH using 74, 112, and 210 compute nodes¹⁸ with a CPU frequency of 2.3 GHz. Table 3 summarizes the PowerDAM tracked history data for those runs.

Number of Compute Nodes	Maximum CPU Frequency (GHz)	Measured EtS Value (kWh)	Measured APC Value (Watt)	Measured TtS Value (min)
74	2.3	9.2	12115	45.58
112	2.3	9.21	18281.16	30.23
210	2.3	9.05	34014.268	15.96

Table 3: PowerDAM tracked EPOCH history data

Assume further, that the user, who is left with a 9.5 kWh energy budget, wants to execute EPOCH using 287 compute nodes¹⁹ and there is a 42 kW system power consumption limit. A possible question to answer could be: *is there a CPU frequency f such that $EtS(287, f) \leq 9.5$ kWh; $APC(287, f) \leq 42$ kW; and $TtS(287, f)$ is minimized?* The usage of LACP will assist in determining the answer to this question.

Figure 9, Figure 10, and Figure 11 show the LACP prediction results correspondingly for EtS, APC, and TtS metrics of EPOCH using the PowerDAM history data presented in Table 3.

First, because of the 42 kW power consumption constraint and the LACP APC prediction results (Figure 10), the allowed maximum CPU frequency must be in [1.2; 2.0] GHz range. Second, because of the leftover 9.5 kWh energy budget, the allowed maximum CPU frequency must be in [1.4; 2.0] GHz range (using the EtS prediction results from Figure 9 and considering the predefined power consumption constraint). Using the LACP TtS prediction results (Figure 11) it can be concluded that the minimal execution time within the allowed [1.4; 2.0] GHz maximum CPU frequency range is achieved using the (287, 2.0) configuration.

Some LACP Prediction Statistics

Hydro: In total, 3 EtS, 1 APC, 5 Transitive TtS, and 9 Direct TtS Hydro predictions out of the 68 measurements showed more than a 10% error, when LACP had available TtS, APC, and EtS information of Hydro for compute nodes 105, 170, 240, and 370 (all executed at maximum CPU frequency of 2.3 GHz).

EPOCH: In total, 1 EtS, 3 APC, 8 Transitive TtS, and 5 Direct TtS EPOCH predictions out of the 106 measurements showed more than a 10% error, when LACP had available TtS, APC, and EtS information of Hydro for 74, 112, and 210

¹⁷The time until the application starts its actual execution.

¹⁸These numbers of compute nodes were randomly selected.

¹⁹This number of compute nodes was randomly selected.

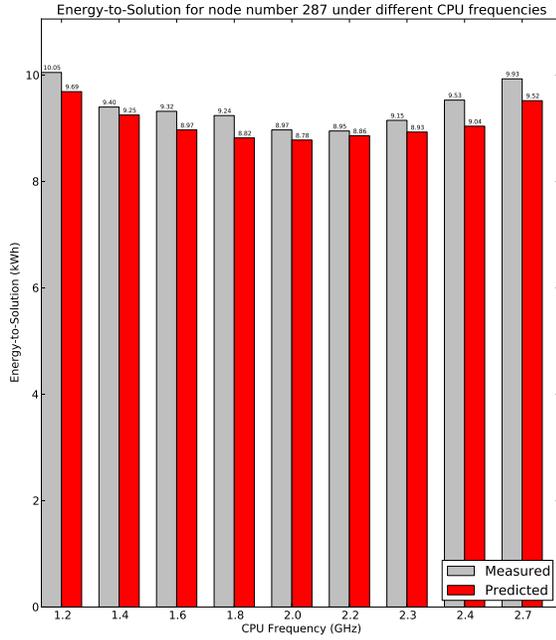


Figure 9: LACP EtS prediction results for 287 compute nodes EPOCH execution under different CPU frequencies
 Note: available data points are for 74, 112, and 210 compute nodes

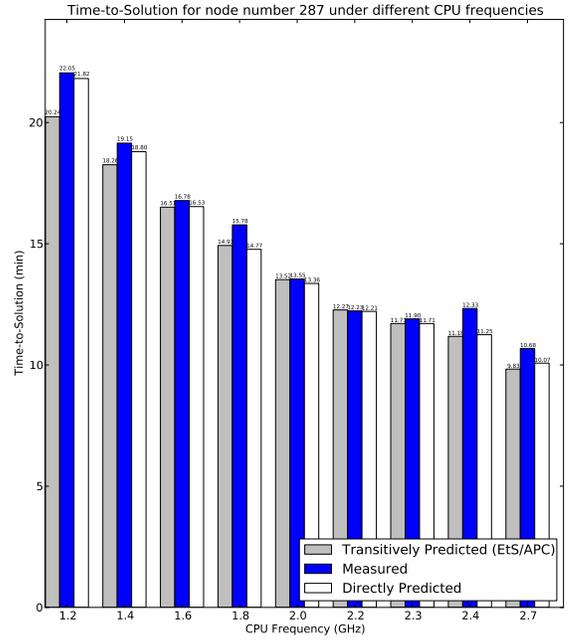


Figure 11: LACP TtS prediction results for 287 compute nodes EPOCH execution under different CPU frequencies
 Note: available data points are for 74, 112, and 210 compute nodes

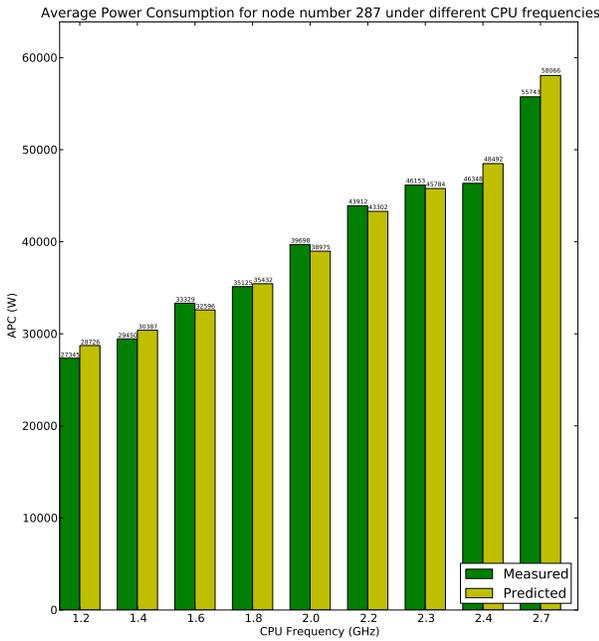


Figure 10: LACP APC prediction results for 287 compute nodes EPOCH execution under different CPU frequencies
 Note: available data points are for 74, 112, and 210 compute nodes

compute nodes (all executed at maximum CPU frequency of 2.3 GHz).

Note that due to the adaptability of the LACP model, the prediction accuracy for a given application is improved with each

additional execution (for a new configuration) of that application.

CONCLUSION AND FUTURE WORK

The analysis and prediction of three application execution relevant metrics, namely: (i) Time-to-Solution (TtS) - the complete execution time; (ii) Energy-to-Solution (EtS) - the aggregated energy consumption; and (iii) Average Power Consumption (APC), were discussed.

The following bullet points summarize the main contributions that have been made to the state of the art:

- the extension of the previously developed AEPCP process to include the processor frequency information;
- the derivation of *three analytical models* for estimating an application's *TtS*, *EtS*, and *APC* as functions of the number of compute nodes and the CPU frequency at which all the cores of all compute nodes operate; and
- the development of the *Lightweight Adaptive Consumption Prediction (LACP)* model implementing the extended AEPCP process and allowing for application TtS, EtS, and APC ahead of time estimation with respect to a given (*compute resource number, maximum CPU frequency*) configuration for a given parallel application.

The following bullet points summarize the main characteristics of the presented LACP model:

- similar to the AEPCP model, LACP prediction is improved with a larger application history TtS/APC/EtS profile achieved, for example, through the additional executions using different configurations;

- the LACP modeling approach is application-neutral, i.e. it does not require any knowledge of application internals;
- the LACP model provides a generic solution that can be used for each application but it produces an application specific result; and
- the prediction accomplished through LACP can be done automatically (if required, transparent from user) for any queued or running set of applications.

In the future, it is planned to apply the model for HPC system, HPC user, HPC data center energy budgeting, and HPC system peak power prediction. The proposed LACP model can also be used to set tighter bounds for the wall-clock time of applications, which in their turn could contribute to the efficiency improvement of the backfilling process of the resource management and scheduling systems.

The shown applicability and the accuracy of the suggested LACP model allow it to be considered as an ideal building block for future energy and power aware resource management systems.

ACKNOWLEDGMENTS

The work presented here has been carried out within the SIMOPEK project which has received funding from the German Federal Ministry of Education and Research (BMBF) under grand agreement no. 01IH13007A. The work was achieved using the GSC (Gauss Center for Supercomputing) resources at BAdW-LRZ with support of the State of Bavaria, Germany.

The EPOCH code used in this research was developed under UK Engineering and Physics Sciences Research Council grants EP/G054940/1, EP/G055165/1 and EP/G056803/1.

The authors would like to thank Jeanette Wilde for her valuable suggestions and comments.

REFERENCES

1. IBM: Tivoli workload scheduler loadleveler. <http://www.ibm.com/systems/software/loadleveler/>, 2015.
2. Amdahl, G. M. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, ACM (1967), 483–485.
3. Arber, T., and et al. EPOCH: Extendable PIC Open Collaboration. <http://ccpforge.cse.rl.ac.uk/gf/project/epoch/>, 2015.
4. Auweter, A., Bode, A., Brehm, M., Brochard, L., Hammer, N., Huber, H., Panda, R., Thomas, F., and Wilde, T. A case study of energy aware scheduling on SuperMUC. In *Supercomputing*, Springer (2014), 394–409.
5. Blaauw, D., Martin, S., MUDGE, T., and Flautner, K. Leakage current reduction in VLSI systems. *Journal of Circuits, Systems, and Computers* 11, 06 (2002), 621–635.
6. Brochard, L., Panda, R., and Vemuganti, S. Optimizing performance and energy of HPC application on POWER7. *Computer Science - Research and Development* 25, 3-4 (2010), 135–140.
7. Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C., et al. *Introduction to algorithms*, vol. 2. MIT press Cambridge, 2001.
8. Dongarra, J. J., Luszczek, P., and Petitet, A. The LINPACK benchmark: Past, present, and future. concurrency and computation: Practice and experience. *Concurrency and Computation: Practice and Experience* 15 (2003), 803-820.
9. Gauss Centre for Supercomputing. <http://www.gauss-centre.eu>, 2015.
10. Ge, R., Feng, X., and Cameron, K. W. Modeling and evaluating energy-performance efficiency of parallel processing on multicore based power aware systems. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, IEEE (2009), 1–8.
11. Humphrey, W. S. *A discipline for software engineering*. Addison-Wesley Longman Publishing Co., Inc., 1995.
12. Kogge, P., Bergman, K., Borkar, S., Campbell, D., Carson, W., Dally, W., Denneau, M., Franzon, P., Harrod, W., Hill, K., et al. Exascale computing study: Technology challenges in achieving exascale systems, 2008.
13. Lavallée, P.-F., de Verdière, G. C., Wautelet, P., Lecas, D., and Dupays, J.-M. Porting and optimizing HYDRO to new platforms and programming paradigms-lessons learnt. http://www.prace-ri.eu/IMG/pdf/porting_and_optimizing_hydro_to_new_platforms.pdf, December 2012.
14. Leibniz Supercomputing Centre (LRZ) of the Bavarian Academy of Sciences, and Humanities. <http://www.lrz.de/>, 2015.
15. Partnership for Advanced Computing in Europe. <http://www.prace-ri.eu/>, 2015.
16. Rauber, T., and Rüniger, G. *Parallel programming: For multicore and cluster systems*. Springer Science & Business, 2013.
17. Rotem, E., Naveh, A., Rajwan, D., Ananthakrishnan, A., and Weissmann, E. Power-management architecture of the intel microarchitecture code-named sandy bridge. *Micro*, IEEE 32, 2 (March 2012), 20–27.
18. Shoukourian, H., Wilde, T., Auweter, A., and Bode, A. Monitoring Power Data: A first step towards a unified energy efficiency evaluation toolset for HPC data centers. *Environmental Modelling & Software* 56 (2013). Thematic issue on Modelling and evaluating the sustainability of smart solutions.

19. Shoukourian, H., Wilde, T., Auweter, A., and Bode, A. Predicting the Energy and Power Consumption of Strong and Weak Scaling HPC Applications. *Supercomputing Frontiers And Innovations* (2014), 20–41.
20. Staff, H. HP power capping and dynamic power capping for ProLiant servers. Tech. rep., HP, Tech. Rep. TC090303TB, 2009.
21. Top500. <http://top500.org/>, 2014.
22. Veendrick, H. J. Short-circuit dissipation of static CMOS circuitry and its impact on the design of buffer circuits. *Solid-State Circuits, IEEE Journal of* 19, 4 (1984), 468–473.
23. Weste, N. H., and Harris, D. M. *CMOS VLSI design: a circuits and systems perspective*. Pearson Education India, 2005.
24. Woo, D. H., and Lee, H.-H. S. Extending Amdahl’s Law for Energy-Efficient Computing in the Many-Core Era. *IEEE computer* 41, 12 (2008), 24–31.

A Virtual Machine Model for Accelerating Relational Database Joins using a General Purpose GPU

Kevin Angstadt

Department of Computer Science
University of Virginia
Charlottesville, VA 22904
kaa2nx@virginia.edu

Ed Harcourt

Department of Computer Science
St. Lawrence University
Canton, NY 13617
edharcourt@stlawu.edu

ABSTRACT

We demonstrate a speedup for database joins using a general purpose graphics processing unit (GPGPU). The technique is novel in that it operates on an SQL virtual machine model developed using CUDA. The implementation compiles an SQL statement to instructions of the virtual machine that are then executed in parallel on the GPU. We use the three-dimensional structure of the CUDA grid and thread model to perform a join on up to three relations at a time. Query execution results in speedups of 2 to 60 times on consumer-level GPUs depending on the size of the result set.

Author Keywords

GPGPU, SQL, virtual machine, join, relational database

ACM Classification Keywords

D.1.3 Concurrent Programming: Parallel Programming;
H.2.4 Database Management: Parallel Databases

INTRODUCTION

A GPU, the major computational resource on a graphics card, has a primary role of computing and rendering images on a computer monitor. The massively parallel architecture of these chips has also been harnessed by researchers and applied to many computationally intense problems. Because GPUs are found in almost all modern computers, moving data processing to a host computer's GPU is a cost-effective method for decreasing execution time. While modern, consumer multi-core CPUs are designed with four to twelve simultaneously-executing threads, current, consumer-level GPUs can execute thousands of threads simultaneously. Such computational power is created at the expense of independent thread execution and large caches; however, the GPU's SIMD architecture is ideal for data processing that executes the same, independent calculation over a very large data set.

GPU manufacturers, such as AMD and NVIDIA have released APIs for their hardware, and the OpenCL framework also allows for the programming of GPUs. NVIDIA's API

and framework, together known as CUDA [11] (Compute Unified Device Architecture), provides extensions to C/C++ as well as a programming interface for utilizing the massively parallel GPU. CUDA allows the programmer to control the memory spaces of both the host device (CPU) and the GPU, inter-thread communication, and mapping of threads and thread blocks to GPU hardware.

Relational database queries often fit a SIMD execution pattern; for example, the operations in a *where*-clause are mapped to every row in a table. A database *join* occurs when two or more source tables are combined into one resulting table based upon pre-defined and static relationships between rows and columns in the tables. The Structured Query Language (SQL), used by most relational database management systems (RDBMS), defines several different types of joins, but the most generic is the *cross-join*. Other joins can be defined in terms of the cross-join. Mathematically, the cross-join of two or more tables computes the Cartesian product (or cross-product) of the rows from each table (*e.g.*, with two tables all possible pairs of rows, three tables all possible triples of rows, etc.). The entire cross product is rarely meaningful. Predicates on the rows of the resulting cross product then restrict the resulting rows to those of interest. Such computations are simple to represent in SQL but are extremely compute-intensive.

The scope of this research is to extend a GPU-based SQL virtual machine to allow for the execution of SQL statements containing joins and to demonstrate the efficacy of such execution methods. While most GPU-based database implementations utilize various parallel primitives, our implementation instead executes queries on an SQL virtual machine. The virtual machine executes on either the host CPU or a CUDA capable GPU allowing for performance comparisons.

RELATED WORK

Using GPUs to accelerate computations is well-established [10] and has a history of being applied to database operations [4, 6, 7]. Our work differs in that it uses an SQL virtual machine to represent the parallelism in the cross join and to map it directly to the geometry of the GPU.

This paper builds directly on the work presented in [1, 2, 3] by adding the join operation, which was not part of the original framework, to the Virginian database. In its initial form, this database was built directly on the SQLite Virtual Machine [12]; however, the current virtual machine implementation,

though inspired by SQLite, is completely custom. The virtual machine represents a compilation target for SQL that can then be executed on either a host CPU or a GPU. The authors show significant speedup in non-join related SQL queries. The Virginian project introduced two new SQL virtual machine instructions for parallel processing, the `Parallel` and `Converge` instructions, to denote the instructions to execute on the GPU. We extend these instructions further for our GPU implementation to allow for parallel joins.

The work [5, 9, 8] includes join queries on GPUs by relying on a set of parallel primitives and produce speedups of 2 to 27 times. A primitive is a function implemented directly as an independent CUDA kernel such as *sort*, *map*, and *filter*. The authors then implement and evaluate various algorithms for joining tables in terms of these primitives including nested loop joins, sort-merge joins, and hash joins. Our work differs in that we implement an SQL virtual machine rather than individual primitives. SQL statements are compiled directly to a virtual machine opcode model rather than represented in terms of higher level CUDA kernel primitives. The benefits of executing a virtual machine as a kernel rather than parallel primitives is thoroughly explored in [2].

IMPLEMENTATION

Consider tables T_1 and T_2 that share an attribute c_2 . The SQL query below computes a cross-join with a predicate that yields the rows in the cross product where the values of c_2 in each table are equal. This SQL statement is the equivalent of the *natural-join*.

```
SELECT * FROM T1, T2 WHERE T1.c2 = T2.c2
```

For example, the cross join of tables T_1 and T_2 in Figure 1 with the predicate $T_1.c_2 = T_2.c_2$ (denoted $T_1 \bowtie T_2$) contains nine rows but the predicate restricts the result table to just the three highlighted rows.

A join such as this can be implemented through *nested loops*, where each loop iterates over an input table and emits rows matching the predicate. Such an implementation is used by SQLite [12]. We adopt this algorithm for our virtual machine as well and use the three-dimensional CUDA thread topology to implement the loop nesting directly.

SQL to Opcode Translation

The implementation parses an SQL query and generates an *abstract syntax tree* (AST) for the query. The AST is then processed in several passes to generate a virtual machine program that represents the query.

Consider the following SQL join query, (part of our benchmarking queries listed in the appendix):

```
SELECT test.id, test1.uniformi,  
test.normali5 FROM  
test, test1 WHERE  
test1.uniformi > 60 AND  
test.normali5 < 0
```

Here, the two tables `test` and `test1` each have six columns: the first three are integer columns and the last three

T_1		\bowtie	T_2		$=$	$T_1 \bowtie T_2$			
c ₁	c ₂		c ₂	c ₃		c ₁	c ₂	c ₂	c ₃
1	w		x	5		1	w	x	5
2	z		z	6		2	z	x	5
3	z		w	7		2	z	w	7
						3	z	x	5
						3	z	z	6
						3	z	w	7

Figure 1: The cross join, $T_1 \bowtie T_2$. The rows in the *natural join* are highlighted.

0: Table	0	0	0	0
1: Table	1	0	1	0
2: ResultColumn	0	0	0	id
3: ResultColumn	0	0	0	uniformi
4: ResultColumn	0	0	0	normali5
5: Parallel	0	0	16	0
6: Column	3	0	1	0
7: Integer	0	60	0	0
8: Le	3	0	14	0
9: Column	4	1	0	0
10: Integer	1	0	0	0
11: Lt	4	1	13	1
12: Invalid	0	0	0	0
13: Rowid	2	0	0	0
14: Result	2	3	0	0
15: Converge	0	0	0	0
16: Finish	0	0	0	0

Figure 2: A sample virtual machine program generated by the SQL compiler.

are floating-point. The query above computes the cross product of the two tables restricting the result table so that the column `test1.uniformi` is greater than 60 and the column `test.normali5` is negative. The resulting virtual machine program is shown in Figure 2.

The virtual machine instructions are explained in [1]; however, this instruction set only operates over a single source table. We therefore extend this to support multiple tables using *table cursors*, which are similar to cursors in SQLite. A table cursor is a positive integer that acts as a pointer to the table data associated with a specific instruction. A cursor is assigned to a table through the third parameter of `Table` instruction, which opens a handle to a table. Similarly, we extend the `Column` and `Rowid` instructions, used for accessing data in a table, to read from a specific table via its cursor. More formally, the syntax of these three modified opcodes is:

Table [table id], [], [cursor], []

Column [destination register], [source column], [cursor], []

Rowid [destination register], [], [cursor], []

For the sample opcode program given in Figure 2 and its corresponding SQL statement given previously, lines 0 and 1 open cursors 0 and 1 to tables `test` and `test1`, respectively. Lines 2 through 5 configure the result table and invoke the parallel portion of the SQL query. Line 6 reads and copies column 0 of the row at cursor 1 to register 3. This value is compared with 60 (loaded in line 7) in line 8. This constitutes the first portion of the `WHERE` clause. A similar comparison is then made in lines 9 through 11 for the second portion of the clause. The value of this column is stored in register 4. If the row is still valid after this filtering, we load the primary key for the row at cursor 0 in line 13. We then copy three registers, beginning with register 2 to the result table in line 14. Lines 15 and 16 complete and clean up the query.

These extensions to the syntax of the instructions maintain backwards compatibility with the previous version of the Virginian database because cursors are passed using previously unused parameters in the instructions.

The execution of the virtual machine program is done at two levels. Instructions outside of the `Parallel` and `Converge` boundary initialize the query and are executed on the host CPU. The parallel portion (those between `Parallel` and `Converge`) of the query runs in a separate virtual machine, which can be implemented on the CPU or also on an accelerator (in our case, a CUDA kernel running on the GPU). The inherent looping over the source table data is obfuscated in this section of the opcode program. This is because the individual result rows in a join are independent. Instead of programming the iteration into the virtual machine program, we describe the processing necessary for a *single* data point in the join and allow the virtual machine (implemented as a kernel) to map this efficiently over all data points in the query.

Therefore, the code generated between the `Parallel` (line 5) and `Converge` (line 15) instructions and how that region can be mapped to the three-dimensional CUDA thread topology described later is of particular interest to us. This mapping essentially flattens the nested loop structure used to compute cross products and allows for all data points to be computed in parallel on the GPU, which we describe shortly.

Tablet Management

We present a brief overview of table data management; for a detailed description, see [2]. Database tables are typically stored using a balanced tree data structure (*e.g.*, a BTree). To take advantage of the grid topology of a CUDA-based GPU we instead store subsets of a table known as *tablets*, which partition the overall table vertically. Data within tablets is stored in column-major order, which allows for better data coalescing on GPUs and caching on CPUs. Each table also contains meta-data about its contents as well as space for keys, fixed-width, and variable-width data. To represent an entire table, tablets are organized into a linked list, and the SQL virtual machine processes one tablet at a time. Data in a tablet can be accessed in constant time via a pointer to the start of a column and a row offset. Currently we assume that source tables fit within one tablet (and the join result may span many

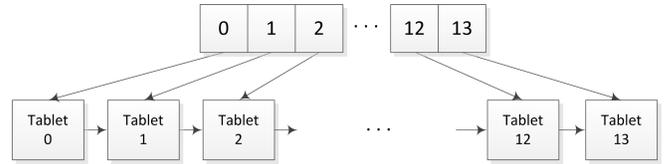


Figure 3: The result table is a linked list of tablets. To allow for constant time access to any row in the table, we maintain an array of pointers to each tablet.

tablets); though, it is straightforward to support larger source tables that span multiple tablets.

When executing a query to join two tables, careful attention must be paid to the resulting table and its tablet representation. The result of a query that involves computing a Cartesian product can consume large amounts of memory. Given a table with m rows and a table with n rows, the length of the table resulting from the cross product can contain as many as $m \cdot n$ rows. Consider the example of crossing two tables, each with 3 500 rows. The result table will contain at most $3\,500 \cdot 3\,500 = 12\,250\,000$ rows. Because there is a limited amount of memory on a GPU, it is possible that this result cannot fit into the available global memory. Consequently, we use *mapped memory* to store both the data and results tablets. Mapped memory is main system memory that has been pinned and mapped via the NVIDIA CUDA API to the graphics card’s memory space. The memory is *pinned* (page-locked) in that it cannot be swapped out by the operating system, and is then guaranteed to be available when accessed from either the host code or the CUDA kernel. By using mapped memory, we can allow for larger joins. One drawback, however, is that memory accesses must now travel across the PCI bus, which is significantly slower than global memory accesses on the GPU. As noted by Bakkum, using mapped memory is still faster than the combined time needed to copy data to and from the GPU as well as executing the code [2].

Suppose that a tablet can hold at most 1 000 000 rows (this number is dependent on the data in the table, but a tablet is no larger than 8MB), and we execute a query that produces the Cartesian product of two tables with 3 500 rows each. The resulting table will span

$$\left\lceil \frac{3\,500 \cdot 3\,500}{1\,000\,000} \right\rceil = \left\lceil \frac{12\,250\,000}{1\,000\,000} \right\rceil = 13 \text{ tablets.}$$

Prior to query execution, we do not know how many rows will be in the result, and so we conservatively allocate all 13 tablets. After query execution, we can delete any unused tablets before returning the result. Because tablets are stored as a linked list, the result table does not provide constant time access to its rows. To retain constant time access, we generate an array of pointers to each tablet in the result table. Instead of walking through the linked list of tablets, we can now directly access each tablet in constant time. Figure 3 shows an example result tablet structure that might be allocated for a query.

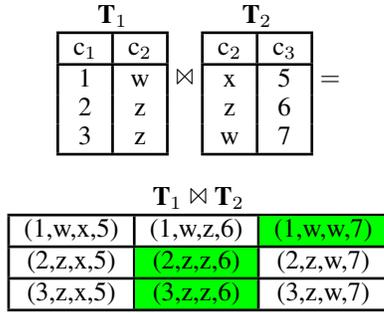


Figure 4: The join of two tables can be constructed in a two-dimensional grid. Highlighted cells indicate rows that remain after filtering on the `WHERE` clause.

Query Execution

CUDA organizes threads into a grid of up to three dimensions. We exploit this 3D topological structure as it coincides nicely with the structure of a Cartesian product where a two table join is a two dimensional grid and a three table join is a three dimensional grid. Consider again the join of two tables T_1 and T_2 from the introduction.

```
SELECT * FROM T1, T2 WHERE T1.c2 = T2.c2
```

An alternative view of the result table is as a two-dimensional CUDA grid as in Figure 4. Here, the row index in the two-dimensional table corresponds to the row index of T_1 while the column index corresponds to the row index of T_2 . We translate this directly to the dimensions of the instantiated CUDA kernel. Thread indices in the x -axis correspond to row indices of the first table in the SQL query, and thread indices in the y -axis correspond similarly to the second table in the query.

Each kernel thread has access to 1) the virtual machine program in constant GPU memory, 2) pointers to the data in the tables it requires as well as 3) pointers to the tablet structure to write back the result rows. For example, in Figure 4 each cell in the result table coincides with a thread and would contain a pointer to the appropriate rows in T_1 and T_2 .

A high-level outline of the process of executing a query is:

1. Copy table meta-data to the GPU
2. Copy source tablets to the GPU
3. Allocate pointers to meta-data on the GPU
4. Launch kernel threads (described below)

A single thread operates on one item in the cross product interpreting the instructions between `Parallel` and `Converge` in the generated opcode program. Recall that these instructions are independent as they describe operations on a single data point in the cross product being evaluated in the query. Therefore, all kernels operate in parallel for all entries in the cross product of the source tables. All threads execute the same exact SQL virtual machine program on the

table rows the thread has been assigned. The kernel implements a function for each virtual machine instruction. These functions run on the GPU only (`__device__` functions) and are called by the kernel threads. Much of the work prior to launching the kernels is in setting up the tablet structure for each thread. Each thread computes the source rows it is responsible for using the built-in CUDA variables `blockIdx`, `blockDim`, and `threadIdx`:

```
row = blockIdx.dim * blockDim.dim + threadIdx.dim
```

where $dim = x$ for the table at cursor 0 and $dim = y$ for the table at cursor 1.

When writing rows to the result table, we must calculate the correct result row indices. Because a thread does not know which other threads will produce rows for the result tablets this requires synchronization between the kernel threads. For all threads within a block that have a valid result row, we *atomically add* 1 to a counter, `block`. Within the CUDA framework, an atomic add returns the current value of `block` to the thread, which we store as `place`. Within a block, each kernel now knows its result row offset. The threads are then synchronized to ensure that all threads have a `place` before proceeding. Next, for each block, we now atomically add `block` to another counter, and threads in each block share the returned value as `start_of_block`. This value indicates the offset in the result table of each block of threads for writing result rows. After another thread synchronization, we are guaranteed that each thread now has a valid `place` and `start_of_block`. The row index in a thread is therefore

$$row_index = start_of_block + place$$

for all threads with a valid result row. With this index, writing to the result table does not require coordination between the threads. From the row index, each thread calculates the tablet and offset of the row relative to a given tablet:

$$tablet_index = \left\lfloor \frac{row_index}{rows_per_tablet} \right\rfloor$$

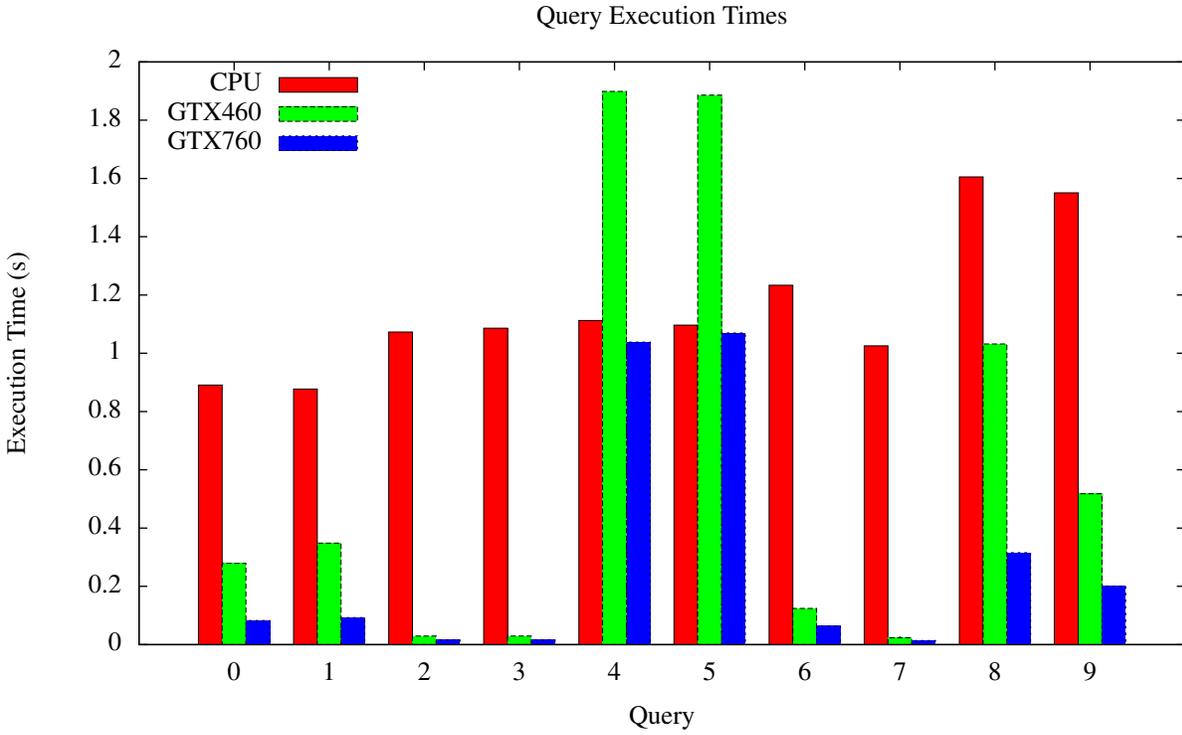
$$offset = row_index - (tablet_index \cdot rows_per_tablet).$$

Using these values, the thread writes its row across the PCI bus to the result table stored in mapped memory.

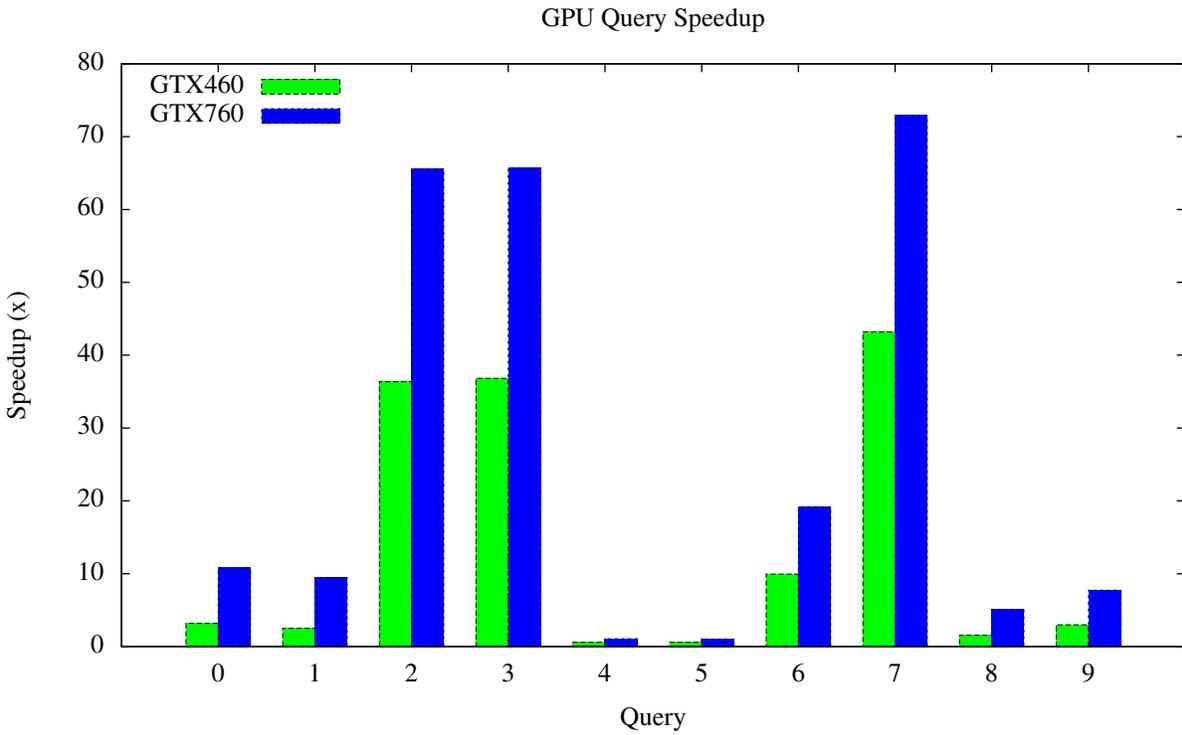
RESULTS

Tests were performed using two 3 500 row tables containing randomly generated values. We use the same source table layout as [2]. Each table consists of an integer primary key, and three columns of values each for both 32-bit integers and IEEE 754 32-bit floating point values. One column is randomly distributed across $[-100, 100]$, the second is a normal distribution with a sigma of 5, and the final column contains a normal distribution with a sigma of 20.

Tests were conducted both on an NVIDIA GTX460 GPU (336 CUDA cores and 1GB memory) and an NVIDIA GTX760 GPU (1152 CUDA cores and 2GB memory) using CUDA 6.5 and the NVIDIA 340.29 driver. The host CPU for



(a)



(b)

Figure 5: Queries show varied execution times (a) and speedup (b) on the GPU for different queries in the test suite.

	CPU	GTX460	GTX760
Integer	1.183	0.673	0.304
Floating Pt.	1.127	0.561	0.279
All	1.155	0.617	0.291

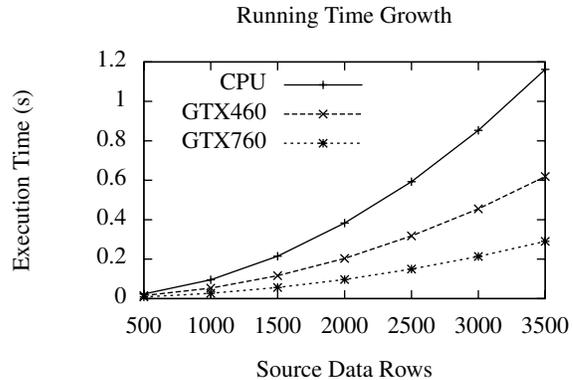
Table 1: Running times in seconds for CPU and GPU execution of integer and floating-point arithmetic queries.

both GPUs was a 2.6 GHz Intel Core i7 920 CPU running the 3.13.0-39-generic Linux kernel. Tests were executed ten times and the data presented here is the average of these runs.

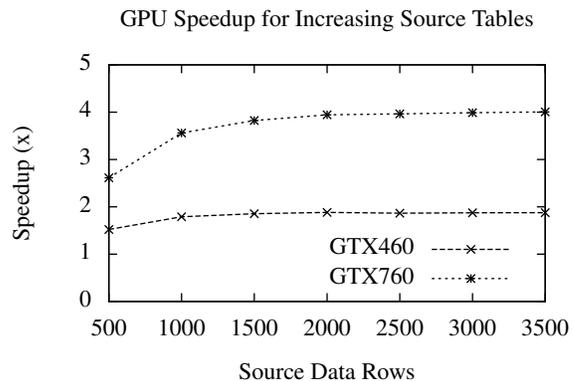
Figure 5 graphically demonstrates the differences in running times on both the CPU and GPU for ten different join queries. The mean execution time for all ten queries on the CPU was 1.155 seconds, and the mean GPU execution time was 0.617 seconds and 0.291 seconds for the GTX460 and GTX760 respectively. Even numbered queries contain predominately integer arithmetic, with each subsequent odd-numbered query executing the same query with floating-point data. Table 1 lists the average running times for each of these two categories on both the GPUs and the CPU. There is no significant difference in values between these integer and floating-point queries, which indicates that speedup is independent of the data type. Additionally, the GPU executed faster than the CPU on average for both tests.

Figure 6(a) depicts the average running time for our suite of ten queries for increasing source table sizes, and figure 6(b) represents this data as speedups. Performance on smaller table sizes is less due to memory writes making up a greater portion of the total execution time. Nevertheless, the GPU implementation of the SQL virtual machine executes approximately twice to four times as fast as the CPU virtual machine on average for this query suite.

Queries 4 and 5 in Figure 5 executed more slowly on the GTX460 than the CPU, and the GTX760 executed in approximately equal time compared with the CPU. We hypothesize that this is due to result table sizes. While all other queries output 2.25 million rows or fewer, these two queries output approximately 6 million rows each. The additional time required to write these results across the PCI bus to the host memory significantly slowed execution time. We then conducted additional benchmarking in order to verify that the memory writes are the limiting step during query execution. To test this hypothesis, we incrementally increase the number of result rows in a cross-join of two, 3 000 row tables. As represented by Figure 7, the GPU becomes less efficient until the GPU executes in the same time as the CPU. For the GTX460, this occurs at approximately 1.8 million rows, and at 4.5 million rows for the GTX760. These values will vary with the computation required by the query and the layout of the desired data in the source tables. The PCI bus was also a limiting factor in our tests because the test machine only supported PCIe2. The GTX760 is designed to utilize the additional throughput of PCIe3, and this additional throughput would push the break-even point even closer to a full Cartesian product.



(a)



(b)

Figure 6: Average performance on the ten query suite for increasing source table sizes. (a) measures running times and (b) measures speedup relative to CPU.

SQL joins that result in a massive number of result rows approaching a full Cartesian product are uncommon and are inherently problematic even in commercial RDBMSs. For more reasonable queries, in which the predicate filters most rows, the GPU remains extremely efficient for varying source table sizes. Figure 8 shows the growth of execution time as a function of source table size for a query with a restrictive predicate. This query limits the result table to be the same size as the input table by joining the two tables on their key, a common join operation in database queries. For this query, speedup ranges from 20 to 30 times on the GTX460 and 40 to 60 times on the GTX760.

CONCLUSIONS AND FUTURE WORK

This paper demonstrates the efficacy of accelerating database joins using an SQL virtual machine based GPU execution. Previous research has shown speedup for VM-based execution for single-table queries on GPUs, and our results indicate that this holds for multiple-table queries as well. Our implementation achieves equal or better speedups as compared with joins implemented with primitives-based kernels.

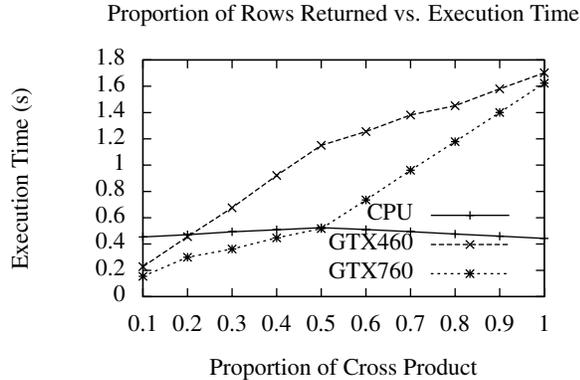


Figure 7: As fewer rows in the Cartesian product are filtered, the GPU becomes less efficient.

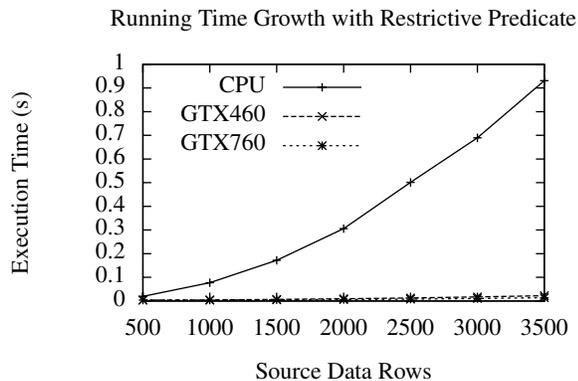


Figure 8: Average running times with restrictive predicate for increasing source table sizes.

Our extensions to the Virginian database system primarily demonstrate the techniques needed to allow for joining multiple tables. The next step would be to ensure that all stages of the virtual machine’s execution are efficiently implemented. Although, we took care to implement a fair computational schema on both the CPU and GPU, kernel memory writes across the PCI bus are currently inefficient. The current method for copying results to the mapped memory space does not take full advantage of coalesced memory accesses, a feature which can be up to an order of magnitude faster than non-coalesced memory accesses [11]. Because memory writes are the limiting step in queries with joins, we anticipate such an implementation to improve the performance of the GPU virtual machine.

Although our framework does not currently support coalesced writes to mapped memory, our multi-dimensional kernel instantiation and use of mapped memory results in an average of 2x-4x speedup over CPU-based query execution. For joins with reasonably restrictive predicates, speedups can be as much as 20x-60x on consumer-level GPUs. Due to the relatively low cost of GPU hardware and its ubiquitous nature in

modern computer systems, a framework such as this provides a low cost alternative to distributed RDBMSs for accelerating query processing.

Another interesting extension to this research would be the dual use of the CPU- and GPU-based virtual machines. In this scenario, pre-processing of the data could help to determine the most efficient virtual machine for query execution. In queries resulting in large amounts of data with relatively little computation, the CPU virtual machine would be selected; otherwise queries would be executed on the GPU. This would help avoid the cases where memory writes limit the overall performance of query execution, but still benefit from GPU speedup in the general case.

The current implementation technique is also limited to joining at most three tables at a time because of the three dimensional nature of CUDA thread blocks. Because joins in SQL are *closed* (the result of joining two tables is another table) in order to join more than three tables a query must be divided into multiple stages. By automating this process, our framework could then theoretically handle an arbitrary number of tables. Additionally, we would like to incorporate other join syntaxes into the SQL parser to allow for additional support of the SQL language. Such extensions allow for simple notation for several types of joins, including `inner`, `outer`, and `natural` joins.

More generally, a natural follow-up to this research would be a study on the scalability of this technique. Our tests use at most 3 500 rows in each source table. Do speedups remain consistent for larger source table sizes, or will tablet management and writes across the PCI bus subsume the overall computation time? Additionally, how does increasing the number of joined tables affect performance?

In addition to improving the efficiency of the software itself, another interesting research path would be multi-card implementations. Splitting data across multiple GPUs has a two-fold advantage: data can be processed more quickly and more data can be processed. Since all table data is stored in mapped memory rather than on the GPU itself, such an implementation could be straight-forward. Each graphics card would be responsible for a different section of the cross product, but each would access the same host memory space. Use of mapped memory also avoids the overhead of copying tables to multiple devices and joining the results back into a single result table after the kernel execution on separate devices completes.

The benchmarks associated with the Virginian framework are highly dependent on the hardware configuration of the test machine; using different hardware may demonstrate different speedups. Compared with more expensive higher performance graphics cards, our hardware was relatively inexpensive with lower performance.

REFERENCES

1. Bakkum, P. The Virginian Database. <https://github.com/bakks/virginian/blob/master/README.md>. Date accessed: February 12, 2015.

2. Bakkum, P., and Chakradhar, S. Efficient data management for GPU databases. Tech. rep., NEC Laboratories America, Princeton, NJ.
3. Bakkum, P., and Skadron, K. Accelerating SQL Database Operations on a GPU with CUDA. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, GPGPU '10*, ACM (New York, NY, USA, 2010), 94–103.
4. Bandi, N., Sun, C., Agrawal, D., and El Abbadi, A. Hardware acceleration in commercial databases: A case study of spatial operations. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30, VLDB '04, VLDB Endowment* (2004), 1021–1032.
5. Fang, R., He, B., Lu, M., Yang, K., Govindaraju, N. K., Luo, Q., and Sander, P. V. GPUQP: Query Co-processing Using Graphics Processors. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, SIGMOD '07*, ACM (New York, NY, USA, 2007), 1061–1063.
6. Govindaraju, N., Gray, J., Kumar, R., and Manocha, D. Gputerasort: High performance graphics co-processor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, SIGMOD '06*, ACM (New York, NY, USA, 2006), 325–336.
7. Govindaraju, N. K., Lloyd, B., Wang, W., Lin, M., and Manocha, D. Fast computation of database operations using graphics processors. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, SIGMOD '04*, ACM (New York, NY, USA, 2004), 215–226.
8. He, B., Lu, M., Yang, K., Fang, R., Govindaraju, N. K., Luo, Q., and Sander, P. V. Relational Query Coprocessing on Graphics Processors. *ACM Trans. Database Syst.* 34, 4 (Dec. 2009), 21:1–21:39.
9. He, B., Yang, K., Fang, R., Lu, M., Govindaraju, N., Luo, Q., and Sander, P. Relational Joins on Graphics Processors. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD '08*, ACM (New York, NY, USA, 2008), 511–524.
10. Kirk, D., and Hwu, W.-m. *Programming Massively Parallel Processors, Second Edition: A Hands-on Approach*. Morgan Kaufmann, 2012.
11. NVIDIA Corporation. NVIDIA CUDA programming guide. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, February 2014. Date accessed: February 12, 2015.
12. SQLite. <http://www.sqlite.org/vdbe.html>. Date accessed: February 12, 2015.

BENCHMARK QUERIES

Listed below are the queries used to evaluate the performance of our SQL virtual machine. These were adapted from [3] and [2].

- 0: **SELECT** test.id, test1.uniformi, test.normali5 **FROM** test, test1 **WHERE** test1.uniformi > 60 **AND** test.normali5 < 0
- 1: **SELECT** test.id, test1.uniformf, test.normalf5 **FROM** test, test1 **WHERE** test1.uniformf > 60.0 **AND** test.normalf5 < 0.0
- 2: **SELECT** test.id, test.uniformi, test1.uniformi **FROM** test, test1 **WHERE** (test.id - test1.id) < 5 **AND** (test.id - test1.id) > -5 **AND** test.uniformi > test1.uniformi
- 3: **SELECT** test.id, test.uniformf, test1.uniformf **FROM** test, test1 **WHERE** (test.id - test1.id) < 5 **AND** (test.id - test1.id) > -5 **AND** test.uniformf > test1.uniformf
- 4: **SELECT** test.id, test1.uniformi, test.normali20 **FROM** test, test1 **WHERE** (test1.uniformi < test.normali20) **AND** (test.normali20 + 40) > (test1.uniformi - 10)
- 5: **SELECT** test.id, test1.uniformf, test.normalf20 **FROM** test, test1 **WHERE** (test1.uniformf < test.normalf20) **AND** (test.normalf20 + 40.0) > (test1.uniformf - 10.0)
- 6: **SELECT** test.id, test.normali5, test1.normali20 **FROM** test, test1 **WHERE** test.normali5 = test1.normali5 **AND** test.normali5 * test1.normali20 >= -5 **AND** test.normali5 * test1.normali20 <= 5
- 7: **SELECT** test.id, test.normalf5, test1.normalf20 **FROM** test, test1 **WHERE** test.normalf5 = test1.normalf5 **AND** test.normalf5 * test1.normalf20 >= -5.0 **AND** test.normalf5 * test1.normalf20 <= 5.0
- 8: **SELECT** test.id, test1.uniformi, test.normali5, test.normali20 **FROM** test, test1 **WHERE** test1.uniformi >= -1 **AND** test1.uniformi <= 1 **OR** test.normali5 >= -1 **AND** test.normali5 <= 1 **OR** test.normali20 >= -1 **AND** test.normali20 <= 1
- 9: **SELECT** test.id, test1.uniformf, test.normalf5, test.normalf20 **FROM** test, test1 **WHERE** test1.uniformf >= -1.0 **AND** test1.uniformf <= 1.0 **OR** test.normalf5 >= -1.0 **AND** test.normalf5 <= 1.0 **OR** test.normalf20 >= -1.0 **AND** test.normalf20 <= 1.0

Performance Analysis and Design of a Hessenberg Reduction using Stabilized Blocked Elementary Transformations for New Architectures

Khairul Kabir
University of Tennessee
kkabir@vols.utk.edu

Azzam Haidar
University of Tennessee
haidar@icl.utk.edu

Stanimire Tomov
University of Tennessee
tomov@icl.utk.edu

Jack Dongarra
University of Tennessee
Oak Ridge National
Laboratory
University of Manchester
dongarra@icl.utk.edu

ABSTRACT

The solution of nonsymmetric eigenvalue problems, $Ax = \lambda x$, can be accelerated substantially by first reducing A to an upper Hessenberg matrix H that has the same eigenvalues as A . This can be done using Householder orthogonal transformations, which is a well established standard, or stabilized elementary transformations. The latter approach, although having half the flops of the former, has been used less in practice, e.g., on computer architectures with well developed hierarchical memories, because of its memory-bound operations and the complexity in stabilizing it. In this paper we revisit the stabilized elementary transformations approach in the context of new architectures – both multicore CPUs and Xeon Phi coprocessors. We derive for a first time a blocking version of the algorithm. The blocked version reduces the memory-bound operations and we analyze its performance. A performance model is developed that shows the limitations of both approaches. The competitiveness of using stabilized elementary transformations has been quantified, highlighting that it can be 20 to 30% faster on current high-end multicore CPUs and Xeon Phi coprocessors.

ACM Classification Keywords

G.1.3 Numerical Analysis: Numerical Linear Algebra—*Eigenvalues and eigenvectors*

Author Keywords

Eigenvalues problem, Hessenberg reduction, Stabilized Elementary Transformations, Multi/Many-core

INTRODUCTION

Eigenvalue problems are fundamental for many engineering and physics applications. For example, image processing, compression, facial recognition, vibrational analysis of mechanical structures, and computing energy levels of electrons in nanostructure materials can all be expressed as eigenvalue problems. The solution of these problems, in particular for nonsymmetric matrices that is of interest to this work, can be accelerated substantially by first reducing the matrix at hand to an upper Hessenberg matrix that has the same eigenvalues as the original one (see Section). This can be done in several ways, e.g., using Householder transformations, which is a well established standard, through elementary orthogonal transformations, or stabilized elementary transformations. The latter approach, although having half the flops of the Householder Hessenberg, has been used less in practice because of its memory-bound operations and the complexity in stabilizing it. In this paper we revisit the stabilized elementary transformations approach in the context of multicore CPUs and Xeon Phi coprocessors.

The reduction approach can be used for other two sided-factorizations as well, e.g., in the tridiagonal reduction algorithm for symmetric eigenvalue problems, or in the bidiagonal reduction for singular value decomposition problems. Besides applications based on the nonsymmetric eigenvalue problem, the Hessenberg reduction is applicable to other areas that exploit for example the fact that the powering of a Hessenberg matrix and solving a Hessenberg system of equations is cheap compared to corresponding algorithms for general matrices[22].

It is challenging to accelerate the two-sided factorizations on new architectures because they are rich in Level 2 BLAS operations, which are bandwidth limited and therefore do not scale on multicore architectures and run only at a fraction of the machine's peak performance. There are techniques that can replace Level 2 BLAS operations with Level 3 BLAS. For example, in factorizations like LU, QR, and Cholesky, the application of consecutive Level 2 BLAS operations that oc-

cur in the algorithms can be delayed and accumulated so that at a later moment the accumulated transformation be applied at once as a Level 3 BLAS (see LAPACK [1]). This approach totally removes Level 2 BLAS from Cholesky, and reduces its amount to $O(n^2)$ in LU, and QR, thus making it asymptotically insignificant compared to the total $O(n^3)$ amount of operations for these factorizations. The same technique can be applied to the Hessenberg reduction based on orthogonal transformations[6], but in contrast to the one-sided factorizations, it still leaves about 20% of the total number of operations as Level 2 BLAS. In practice, these 20% of Level 2 BLAS do not scale well on current architectures and dominate the total execution time. Therefore, a very important aspect in enabling the Hessenberg reduction using stabilized elementary transformations to run efficiently on new architectures, is to what extent blocking can be applied, and what is the number of flops remaining in Level 2 BLAS.

Besides the algorithmic and performance modeling aspects related to the importance of reducing the Level 2 BLAS flops, this work is also focused on the computational challenges of developing high-performance routines for new architectures. We describe a number of optimizations that lead to performance as high as 95% of the theoretical/model peak for multicore CPUs and 80% of the model peak for Intel Xeon Phi coprocessors. These numbers are indicative for a high level of optimization achieved – note that the use of accelerators is known to achieve smaller fraction of the peak compared to non-accelerated systems, e.g., for the Top500 HPL benchmark for GPU/MIC-based supercomputers this is about 60% of the peak, and for LU on single coprocessor is about 70% of the peak [4].

BACKGROUND

The eigenvalue problem is to find an eigenvector x and eigenvalue λ that satisfy

$$Ax = \lambda x,$$

where A is a symmetric or nonsymmetric $n \times n$ matrix. When the full eigenvalue decomposition is computed we have

$$A = X\Lambda X^{-1},$$

where Λ is a diagonal matrix of the eigenvalues and X is a matrix of the eigenvectors of A .

In general, solving the eigenvalue problem can be split into three main phases:

1. **Reduction phase:** orthogonal matrices Q are applied on both the left and the right side of A to reduce it to a condensed form matrix – hence these are called “two-sided factorizations.” Note that the use of two-sided orthogonal transformations guarantees that A has the same eigenvalues as the reduced matrix, and the eigenvectors of A can be easily derived from those of the reduced matrix (step 3);
2. **Solution phase:** an eigenvalue solver further computes the eigenpairs Λ and Z of the condensed form matrix;
3. **Back transformation phase:** if required, the eigenvectors of A are computed by multiplying Z by the orthogonal matrices used in the reduction phase.

In this paper we are interested in the nonsymmetric eigenvalue problem. Thus, the reduction phase reduces the nonsymmetric matrix A to an upper Hessenberg form,

$$H = Q^T A Q.$$

For the second phase, QR iteration is used to find the eigenpairs of the reduced Hessenberg matrix H by further reducing it to (quasi) upper triangular Schur form, $S = E^T H E$. Since S is in a (quasi) upper triangular form, its eigenvalues are on its diagonal and its eigenvectors Z can be easily derived. Thus, A can be expressed as:

$$A = Q H Q^T = Q E S E^T Q^T,$$

which reveals that the eigenvalues of A are those of S , and the eigenvectors Z of S can be back-transformed to eigenvectors of A as $X = Q E Z$.

There are many ways to formulate mathematically and solve these problems numerically, but in all cases, designing an efficient computation is challenging because of the nature of the algorithms. In particular, the orthogonal transformations applied to the matrix are two-sided, i.e., transformations are applied on both the left and right side of the matrix. This creates data dependencies that prevent the use of standard techniques to increase the computational intensity of the computation, such as blocking and look-ahead, which are used extensively in the one-sided LU, QR, and Cholesky factorizations. Thus, the reduction phase can take a large portion of the overall time, and it is very important to identify its bottlenecks. The classical approach (LAPACK algorithms `dgehrd`) to reduce a matrix to Hessenberg form is to use the Householder reflectors [24]. The computational complexity of this procedure is about $\frac{10n^3}{3}$.

The reduction to Hessenberg, besides the use of orthogonal transformations based on Householder reflectors, may also be achieved in a stable manner by either stabilized elementary matrices or elementary unitary matrices [18, 12]. This later approach reduces the computational cost by half. In this paper we study and focus on the reduction to Hessenberg using elementary matrices. We revisit the algorithm as well as we accelerate it by implementing a blocked version on both multicore CPUs and Xeon Phi coprocessors. We also revisited the use of elementary matrices to reduce the general matrix to tridiagonal form as described in [8].

Note that in this approach the transformations used in the reduction phase are not orthogonal anymore as explained for the general case at the beginning, and therefore Q^T is replaced by the inverse of the elementary transformation at hand, so that the reduced and the original matrix still have the same eigenvalues (see next).

RELATED WORK

The earliest standard method for computing the eigenvalues of a dense nonsymmetric matrix is based on the QR iteration algorithm [7]. This schema is prohibitively expensive compared to a two phases scheme that first reduces the matrix to Hessenberg form (using either elementary or orthogonal similarity transformations), and then uses a few QR iterations

to compute the eigenvalues of the reduced matrix. This two phase approach using Householder reflectors [24] was implemented in the standard EISPACK software [5]. Blocking was introduced in LAPACK, where a product of Householder reflectors $H_i = I - \tau_i v_i v_i^T$, $i = 1, \dots, nb$ were grouped together using the so called *compact WY transform* [2, 20]:

$$H_1 H_2 \dots H_{nb} \equiv I - V T V^T,$$

where nb is the blocking size, $V = (v_1 | \dots | v_{nb})$, and T is $nb \times nb$ upper triangular matrix.

Alternatively to the Householder reflector approach, the use of stabilized elementary matrices for the Hessenberg reduction has been well known [18]. Later [8] proposed a new variant that reduce the general matrix further to tridiagonal form. The main motivation was that iterating with a tridiagonal form is attractive and extremely beneficial for non symmetric matrices. However, there was two major difficulty here, first is that the QR iteration does not maintain the tridiagonal form of a nonsymmetric matrix and second reducing the nonsymmetric matrix to tridiagonal by similarity transformations encounter stability and numerical issues. To overcome the first issue, [19] proposed the LR iteration algorithm which preserves the tridiagonal form. [8] proposed some recovery techniques in his paper and later [12, 23] proposed another variant that reduce the nonsymmetric matrix to a similar banded form and [13] provided an error analysis of its BHES algorithm. Up to our knowledge, blocking to the stabilized elementary reduction is introduced in this paper, similar to the blocking for the one-sided LU, QR and Cholesky factorizations (see Section).

A hybrid Hessenberg reduction that uses both multicore CPUs and GPUs was introduced first through the MAGMA library [21]. The critical for the performance Level 2 BLAS were offloaded for execution to the high-bandwidth GPU and proper data mapping and task scheduling was applied to reduce CPU-to-GPU communications.

Recent algorithmic work on the two-sided factorizations has been concentrated on two- (or more) stage approaches. In contrast to the standard approach from LAPACK that uses a ‘‘single stage’’, the new ones first reduce the matrix to band form, and second, to the final form, e.g., tridiagonal for symmetric matrices. One of the first uses of a two-step reduction occurred in the context of out-of-core solvers for generalized symmetric eigenvalue problems [9], where a multi-stage method reduced a matrix to tridiagonal, bidiagonal, and Hessenberg forms [17]. With this approach, it was possible to recast the expensive memory-bound operations that occur during the panel factorization into a compute-bound procedure. Consequently, a framework called Successive Band Reductions (SBR) was created [3]. A multi-stage approach has also been applied to the Hessenberg reduction [16] as well as the QZ algorithm [15] for the generalized non-symmetric eigenvalue problem. These approaches were also developed for hybrid GPU-CPU systems [11].

ALGORITHMIC ADVANCEMENTS

In this section we describe the Hessenberg reduction algorithm that uses stabilized elementary matrices. A nonsymmetric $n \times n$ matrix A is reduced to upper Hessenberg form H by stabilized elementary matrices in $n - 2$ steps. At step k the original matrix $A_1 \equiv A$ is reduced to A_{k+1} which is in upper Hessenberg form in its first k columns. Applying elementary transformation matrix L_{k+1} from the right, and then L_{k+1}^{-1} from the left to A_k introduces zeros below the first subdiagonal of column k and generates A_{k+1} by updating columns $k + 1, \dots, n$ of A_k . The algorithm is performed in-place where A_{k+1} overwrites A_k and elementary transformation matrix L_{k+1} can be stored in A_{k+1} . The relationship between A_{k+1} and A_k is expressed as,

$$A_{k+1} = L_{k+1}^{-1} A_k L_{k+1}. \quad (1)$$

The elementary transformation matrix L_{k+1} is defined as

$$L_{k+1} \equiv (I + l_{k+1} e_{k+1}^*),$$

where $l_{k+1} = [0, \dots, 0, l_{k+2}, \dots, l_n]^T$ with $l_i = A_{i,k}/A_{k+1,k}$ for $i = k + 2, \dots, n$, and $e_{k+1}^* = [0, \dots, p_{k+1}, 0, \dots, 0]$ with $p_{k+1} = 1$. The inverse of L_{k+1} is

$$L_{k+1}^{-1} = (I - l_{k+1} e_{k+1}^*),$$

as one can easily check that indeed $L_{k+1} L_{k+1}^{-1} = I$. Certain permutations can be introduced to stabilize the reduction, leading to the following reformulation of equation (1):

$$A_{k+1} = L_{k+1}^{-1} P_{k+1} A_k P_{k+1} L_{k+1}. \quad (2)$$

Here P_{k+1} is an elementary permutation matrix. For simplicity of the explanation, we can ignore the permutation matrix for the rest of the analysis. Namely, we can rewrite (1) as:

$$\begin{aligned} A_{k+1} &= L_{k+1}^{-1} A_k L_{k+1} = L_{k+1}^{-1} \dots L_2^{-1} A_1 L_2 \dots L_{k+1} \\ &= (I - l_{k+1} e_{k+1}^*)(I - l_k e_k^*) \dots (I - l_2 e_2^*) \\ &\quad A_1 (I + l_2 e_2^*)(I + l_3 e_3^*) \dots (I + l_{k+1} e_{k+1}^*) \\ &= (I - l_{k+1} e_{k+1}^*)(I - l_k e_k^*) \dots (I - l_2 e_2^*) \\ &\quad A(I + l_2 e_2^*)(I + l_3 e_3^*) \dots (I + l_{k+1} e_{k+1}^*) \end{aligned} \quad (3)$$

Since $e_k^* l_{k+1} = 0$, $(I + l_2 e_2^*)(I + l_3 e_3^*) \dots (I + l_{k+1} e_{k+1}^*) = (I + l_2 e_2^* + l_3 e_3^* + \dots + l_{k+1} e_{k+1}^*)$. Therefore, (3) becomes:

$$\begin{aligned} A_{k+1} &= (I - l_{k+1} e_{k+1}^*)(I - l_k e_k^*) \dots (I - l_2 e_2^*) \\ &\quad A(I + l_2 e_2^* + l_3 e_3^* + \dots + l_{k+1} e_{k+1}^*) \\ &= (I - l_{k+1} e_{k+1}^*)(I - l_k e_k^*) \dots (I - l_2 e_2^*) \\ &\quad (A + A l_2 e_2^* + A l_3 e_3^* + \dots + A l_{k+1} e_{k+1}^*) \\ &= (I - l_{k+1} e_{k+1}^*)(I - l_k e_k^*) \dots (I - l_2 e_2^*) B = R. \end{aligned} \quad (4)$$

Here $B = (A + A l_2 e_2^* + A l_3 e_3^* + \dots + A l_{k+1} e_{k+1}^*)$ and $R = (I - l_{k+1} e_{k+1}^*)(I - l_k e_k^*) \dots (I - l_2 e_2^*) B$. Let $L_{2:(k+1)}$ be the product of the elementary transformations $L_2 L_3 \dots L_{k+1}$. Then,

$$R = (I - l_{k+1} e_{k+1}^*)(I - l_k e_k^*) \dots (I - l_2 e_2^*) B$$

is written as,

$$\begin{aligned} B &= (I + l_2 e_2^*)(I + l_3 e_3^*) \dots (I + l_{k+1} e_{k+1}^*) R \\ &= (I + l_2 e_2^* + l_3 e_3^* + \dots + l_{k+1} e_{k+1}^*) R = L_{2:(k+1)} R. \end{aligned} \quad (5)$$

Based on (4) and (5), we can derive the blocked version of the reduction algorithm. Now the product of elementary transformation matrices, $L_{2:(k+1)}$ is partitioned as:

$$\begin{aligned} L_{2:(k+1)} &= L_2 L_3 \dots L_{k+1} \\ &= (I + l_2 e_2^*)(I + l_3 e_3^*) \dots (I + l_{k+1} e_{k+1}^*) \\ &= (I + l_2 e_2^* + l_3 e_3^* + \dots + l_{k+1} e_{k+1}^*) \\ &= \begin{pmatrix} 1 & 0 & 0 & \dots & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 & 0 & \dots & 0 \\ 0 & l_{3,2} & 1 & \dots & 0 & 0 & \dots & 0 \\ 0 & l_{4,2} & l_{4,3} & \dots & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & l_{k+1,2} & l_{k+1,3} & \dots & 1 & 0 & \dots & 0 \\ 0 & l_{k+2,2} & l_{k+2,3} & \dots & l_{k+2,k} & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & l_{n,2} & l_{n,3} & \dots & l_{n,k} & 0 & \dots & 1 \end{pmatrix} \\ &= \begin{bmatrix} L_{11} & 0 \\ L_{21} & I \end{bmatrix} \end{aligned}$$

If we partition B and R matrix as well we can rewrite equation (5) for block matrix as,

$$\begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & I \end{bmatrix} \times \begin{bmatrix} R_{11} & R_{12} \\ R_{21} & R_{22} \end{bmatrix}$$

If $[B_{11} B_{21}]^T$ is in upper Hessenberg form we can update $[R_{12} R_{22}]^T$ as follows,

$$\begin{aligned} L_{11} R_{12} &= B_{12} \\ \text{and } B_{22} &= R_{22} + L_{21} R_{12} \\ \Rightarrow R_{22} &= B_{22} - L_{21} R_{12} \end{aligned}$$

Block R_{12} is computed using triangular solve and block R_{22} is updated using matrix multiplication. After k steps the original matrix A is replaced by matrix R where it is in upper Hessenberg form in its first k column.

$$\begin{aligned} \begin{bmatrix} A_{11}^{(k+1)} & A_{12}^{(k+1)} \\ A_{21}^{(k+1)} & A_{22}^{(k+1)} \end{bmatrix} &= \begin{bmatrix} L_{11} & 0 \\ L_{21} & I \end{bmatrix}^{-1} \begin{bmatrix} A_{11}^{(1)} & A_{12}^{(1)} \\ A_{21}^{(1)} & A_{22}^{(1)} \end{bmatrix} \begin{bmatrix} L_{11} & 0 \\ L_{21} & I \end{bmatrix} \\ &= \begin{bmatrix} R_{11} & R_{12} \\ R_{12} & R_{22} \end{bmatrix} \end{aligned}$$

Here $[R_{11} R_{12}]^T$ is in upper Hessenberg form. To reduce the rest of the matrix, A_{22}^{k+1} we proceed the same way as above and repartition $A^{(k+1)}$ as follows,

$$\begin{bmatrix} A_{11}^{(k+1)} & A_{12}^{(k+1)} & A_{13}^{(k+1)} \\ A_{21}^{(k+1)} & A_{22}^{(k+1)} & A_{23}^{(k+1)} \\ A_{31}^{(k+1)} & A_{32}^{(k+1)} & A_{33}^{(k+1)} \end{bmatrix}.$$

Then, in next k steps, $[A_{22}^{(k+1)} A_{32}^{(k+1)}]^T$ is reduced to upper Hessenberg form and the trailing matrix $[A_{23}^{(k+1)} A_{33}^{(k+1)}]^T$ is

updated in the same way as A_{22}^{k+1} was updated after the first k steps. When we worked on $[A_{22}^{(k+1)} A_{32}^{(k+1)}]^T$, the reduction does not have any impact on $[A_{11}^{(k+1)} A_{21}^{(k+1)} A_{31}^{(k+1)}]^T$. Then, after $2k$ steps, $A = A_1$ is updated by A_{2k+1} as follows,

$$\begin{aligned} \begin{bmatrix} A_{11}^{(1)} & A_{12}^{(1)} & A_{13}^{(1)} \\ A_{21}^{(1)} & A_{22}^{(1)} & A_{23}^{(1)} \\ A_{31}^{(1)} & A_{32}^{(1)} & A_{33}^{(1)} \end{bmatrix} &\rightarrow \begin{bmatrix} A_{11}^{(k+1)} & A_{12}^{(k+1)} & A_{13}^{(k+1)} \\ A_{21}^{(k+1)} & A_{22}^{(k+1)} & A_{23}^{(k+1)} \\ A_{31}^{(k+1)} & A_{32}^{(k+1)} & A_{33}^{(k+1)} \end{bmatrix} \\ &\rightarrow \begin{bmatrix} A_{11}^{(k+1)} & A_{12}^{(k+1)} & A_{13}^{(k+1)} \\ A_{21}^{(k+1)} & A_{22}^{(2k+1)} & A_{23}^{(2k+1)} \\ A_{31}^{(k+1)} & A_{32}^{(2k+1)} & A_{33}^{(2k+1)} \end{bmatrix}. \end{aligned}$$

We have not updated $[A_{12}^{(k+1)} A_{13}^{(k+1)}]$ yet – the application of $L_{(k+2):(2k+1)}^{-1}$ from the left does not impact it, while the application of $L_{(k+2):(2k+1)}$ from the right has the following effect:

$$\begin{aligned} \begin{bmatrix} A_{12}^{(2k+1)} & A_{13}^{(2k+1)} \end{bmatrix} &= \begin{bmatrix} A_{12}^{(k+1)} & A_{13}^{(k+1)} \end{bmatrix} \times \begin{bmatrix} L_{22} & 0 \\ L_{32} & I \end{bmatrix} \\ &= \begin{bmatrix} A_{12}^{(k+1)} L_{22} + A_{13}^{(k+1)} L_{32} & A_{13}^{(k+1)} \end{bmatrix} \end{aligned} \quad (6)$$

To summarize the description, we give the pseudo-code of the reduction for the non-blocked and the blocked version in Algorithm 1 and Algorithm 2, respectively.

Algorithm 1: Non-blocked algorithm of Hessenberg reduction using elementary transformation.

```

for  $k = 1$  to  $n - 2$  by 1 do
    find max in  $A(k+1 : n, k)$  and  $j$  is its index
    Interchange rows  $k+1$  and  $j$ 
    Interchange columns  $k+1$  and  $j$ 
     $A(k+2 : n, k) = A(k+2 : n, k) / A(k+1, k)$  (xscal)
     $A(1 : n, k+1) += A(1 : n, k+2 : n) A(k+2 : n, k)$  (xgemv)
     $A(k+2 : n, k+1 : n) -= A(k+2 : n, k) A(k+1, k+1 : n)$  (xger)

```

OPTIMIZATIONS AND PERFORMANCE ANALYSIS

We benchmark our implementations on an Intel multicore system with two 8-core Intel Xeon E5-2670 (Sandy Bridge) CPUs, running at 2.6 GHz. Each CPU has a 24 MB shared L3 cache, and each core has a private 256 KB L2 and 64 KB L1 caches. The system is equipped with 52 GB of memory. The theoretical peak in double precision is 20.8 Gflop/s per core, giving 332 Gflop/s in total. For the accelerators experiments, we used an Intel Xeon-Phi KNC 7120 coprocessor. It has 15.1 GB, runs at 1.23 GHz, and yields a theoretical double precision peak of 1,208 Gflop/s. We used the MPSS 2.1.5889-16 software stack, the icc compiler that comes with the composer_xe_2013.sp1.2.144 suite, and BLAS implementation from MKL (Math Kernel Library) 11.01.02 [14].

The EISPACK library has routine `elmhes` which computes the Hessenberg reduction using elementary transformations. This routine is a serial, non-blocked implementation, and

Algorithm 2: Blocked algorithm of Hessenberg reduction using elementary transformation.

```

for k = 1 to n - 2 by nb do
  nb = min(nb, n - k - 1)
  for i = 1 to nb by 1 do
    if i > 1 then
      A(k + 1 : k + i - 1, k + i - 1) = L(1 : i - 1, 1 :
        i - 1)-1A(k + 1 : k + i - 1, k + i - 1) (xtrsv)
      A(k + i : n, k + i - 1) = A(k + i : n, k :
        k + i - 2)A(k + 1 : k + i - 1, k + i - 1) (xgemv)
      find max in A(k + i : n, k + i - 1) and j is its index
      Interchange rows k + i & j, columns k + i & j
      A(k + i + 1 : n, k + i - 1) / = A(k + i, k + i - 1) (xscal)
      L(i, i) = 1
      L(i + 1 : n - k - i, i) = A(k + i + 1 : n, k + i - 1)
      A(k : n, k + i) + = A(k : n, k + i + 1 : n)A(k + i + 1 : n, k + i - 1)
      (xgemv)
    if k > 1 then
      A(1 : k, k + 1 : k + nb) = A(1 : k, k + 1 : n)L(1 : n - k, 1 : nb)
      (xgemm1)
      A(k + 1 : k + nb, k + nb : n) = L(1 : nb, 1 : nb)-1A(k + 1 :
        k + nb, k + nb : n) (xtrsm)
      A(k + nb + 1 : n, k + nb : n) = A(k + nb + 1 : n, k :
        k + nb - 1)A(k + 1 : k + nb, k + nb : n) (xgemm2)

```

thus depends on the performance of the Level 2 BLAS operations. We implemented a similar serial version since we realized that the EISPACK version unrolls the BLAS operations and this might slow the code. Our serial version is about 3 – 5× faster, as shown in Figure 1. This is due to the fact that we use optimized `xgemv` and `xger` kernels from the MKL library. In order to evaluate its performance in both serial and parallel, we developed a parallel nonblocked version as described in Algorithm 1. We illustrate in Figure 1 a comparison of the performance of the EISPACK routine (`EISPACK_Elementary_nonblocked_CPU`) and our nonblocked version (`MAGMA_Elementary_nonblocked_CPU`). As expected, both performances are limited by the performance of the Level 2 BLAS operations. Even if our nonblocked implementation is parallel (here using 16 threads), we can observe only around 4× speedup. This is due to the fact that the parallel Level 2 BLAS performance is limited by the memory bandwidth. As consequence, it is clear that further significant improvements can be obtained only through algorithmic redesign, as in the blocked version of the algorithm where we replace as many as possible Level 2 BLAS operations (`xger`) by Level 3 BLAS (`xtrmm`, `xtrsm`, `xgemm`). The performance of `dgemm` on our testing machine is around 20× higher than the performance of `dger`, and since around 60% of the flops are in `xgemm`, we expect a maximum of 2.5× speedup in double precision over the nonblocked version. Figure 1 shows the performance obtained from the parallel blocked implementation (`MAGMA_Elementary_blocked_CPU`), as described in Algorithm 2. Our blocking factor n_b is equal to 64. The gain obtained here is around 2.3×, which corresponds to our expectation. However, even though 60% of the flops are now in Level 3 BLAS, the overall performance in Gflop/s obtained (around 35 Gflop/s) remains low compared to the capability of the machine (> 300 Gflop/s).

Performance Bound Analysis

In order to evaluate if the obtained performance results are acceptable and to analyse if there is an opportunity to more improvement as well as to compare the cost of this approach to the classical Householder technique, we conducted a computational analysis of the reduction to Hessenberg using either the elementary or the Householder transformations. For simplicity we show the cost of double precision implementation but it is easily to derive the other precision.

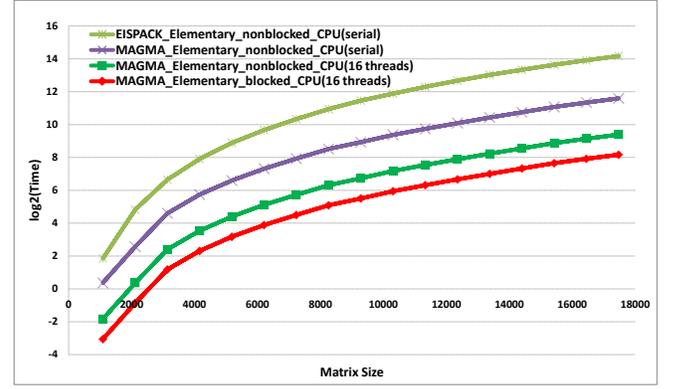


Figure 1. Performance of Hessenberg reduction using elementary transformation on CPU

Similar to the one-sided factorizations (Cholesky, LU, QR), the two-sided reduction to Hessenberg (either using Householder or Elementary) is split into a *panel factorization* and a *trailing matrix update*. Unlike the one-sided factorizations, the panel factorization requires computing Level 2 BLAS matrix-vector product (`xgemv`) involving the entire trailing matrix. This requires loading the entire trailing matrix into memory incurring a significant amount of memory bound operations. This creates data dependencies and produces artificial synchronization points between the panel factorization and the trailing submatrix update steps that prevent the use of standard techniques to increase the computational intensity of the computation, such as look-ahead, which are used extensively in the one-sided LU, QR, and Cholesky factorizations. Let us compute the cost of the algorithm. The blocked implementation of the reduction proceeds by steps of size n_b where the cost of each step consists of the cost of the panel and the cost of the update.

- The panel is of size n_b columns. The factorization of every column involves one matrix-vector product (`xgemv`) with the trailing matrix that constitutes 95% of its operations. Thus the cost of a panel is $n_b \times 2l^2 + \Theta(n)$. Note that l is the size of the matrix at a step i . For simplicity, we omit $\Theta(n)$ and round the cost of the panel by the cost of the matrix-vector product.
- The update of the trailing matrix consists of applying either the Householder reflectors or the Elementary matrices generated during the panel factorization to the trailing matrix from both the left and the right side.

For Householder: The update follows three steps, first and second are the application from the right and third is the application from left:

- 1- $A_{1:n,i+n_b:n} \leftarrow A_{1:n,i+n_b:n} - Y \times V^T$ using `dgemm`,
 - 2- $A_{1:i,i+1:i+n_b} \leftarrow A_{1:i,i+1:i+n_b} - Y_{1:i} \times V^T$ using `dtrmm`,
 - 3- $A_{i:n,i+n_b:n} \leftarrow A_{i:n,i+n_b:n} (I - V \times T^T V^T)$ using `dlarf`.
- Its cost is $2n_b k n + n_b i^2 + 4n_b k (k + n_b)$ flops where $k = n - i - n_b$ is the size of the trailing matrix at a step i . Note that V , T , and Y are generated by the panel phase.

For Elementary: The update follows three steps:

- 1- $A_{1:i,i+1:i+n_b} \leftarrow A_{1:i,i+1:n} \times A_{i:n,i+1:i+n_b}$ using `dgemm`,
 - 2- $A_{i+1:i+n_b,i+n_b:n} \leftarrow A_{i+1:i+n_b,i+1:i+n_b}^{-1} \times A_{i+1:i+n_b,i+n_b:n}$ using `dtrsm`,
 - 3- $A_{i+n_b:n,i+n_b:n} \leftarrow A_{i+n_b:n,i+n_b:n} - A_{i+n_b:n,i+1:i+n_b} \times A_{i+1:i+n_b,i+n_b:n}$ using `dgemm`.
- Its cost is $2n_b i (n - i) + n_b^2 k + 2n_b k^2$ flops.

For all steps (n/n_b), the trailing matrix size varies from n to n_b by steps of n_b , where l varies from n to n_b and k varies from $(n - n_b)$ to $2n_b$. Thus the total cost for the n/n_b steps is:

For Householder:

$$\begin{aligned}
&= 2n_b \sum_{n_b}^{n/n_b} l^2 + 2n_b n \sum_{2n_b}^{n/n_b} k + n_b \sum_{n_b}^{n/n_b} i^2 + 4n_b n \sum_{2n_b}^{n/n_b} k(k + n_b) \\
&= \frac{2}{3} n^3_{\text{dgemv}} + n^3_{\text{Level 3}} + \frac{1}{3} n^3_{\text{Level 3}} + \frac{4}{3} n^3_{\text{Level 3}} \\
&= \frac{10}{3} n^3 \text{ flops.} \tag{7}
\end{aligned}$$

For Elementary:

$$\begin{aligned}
&= 2n_b \sum_{n_b}^{n/n_b} l^2 + 2n_b \sum_{n_b}^{n/n_b} i(n - i) + n_b^2 \sum_{n_b}^{n/n_b} k + 2n_b n \sum_{2n_b}^{n/n_b} k^2 \\
&= \frac{2}{3} n^3_{\text{dgemv}} + \frac{1}{3} n^3_{\text{Level 3}} + \Theta(n^2) + \frac{2}{3} n^3_{\text{Level 3}} \\
&= \frac{5}{3} n^3 \text{ flops.} \tag{8}
\end{aligned}$$

The maximum performance P_{max} that the reduction using elementary transformations can achieve is

$$\begin{aligned}
P_{max} &= \frac{\text{number of operations}}{\text{minimum time } t_{min}} \\
&= \frac{\frac{5}{3} n^3}{t_{min}(\frac{2}{3} n^3 \text{ flops in gemv}) + t_{min}(\frac{3}{3} n^3 \text{ flops in Level3})} \\
&= \frac{\frac{5}{3} n^3}{\frac{2}{3} n^3 * \frac{1}{P_{gemv}} + \frac{3}{3} n^3 * \frac{1}{P_{Level3}}} \\
&= \frac{5 * P_{Level3} * P_{gemv}}{2 * P_{Level3} + 3P_{gemv}} \approx \frac{5}{2} * P_{gemv} \text{ when } P_{Level3} \gg P_{gemv}
\end{aligned}$$

Since the Level 3 BLAS operations are considered as compute bound while the Level 2 are considered as memory bound the gap in performance between these level is large enough such a way that allow us to consider that $P_{Level3} \gg P_{gemv}$. In our testing machine, the maximum performance of `dgemv` is about 14 Gflop/s while the performance of Level 3 BLAS is more than 280 Gflop/s. As consequence, the upper bound limit of the performance that the reduction to band algorithm can reach is always less than $2.5 \times$ the performance

of `dgemv`. We shows in Figure 2 the performance of the `dgemv` routine as well as the theoretical upper bound as described above and the performance of our Magma implementation of the Hessenberg reduction on CPU.

Optimization and design for accelerators

It is clear that the performance obtained from our CPU implementation reaches close to its theoretical bound and thus we believe that there is no more room for improvement. For that we decided to take advantage of accelerators that provide higher range of `dgemv` performance and thus we can expect that the reduction can be accelerated. For the Intel Xeon Phi the `dgemv` peak performance is around 39 Gflop/s while the `dgemm` is more than 800 Gflop/s. Since the `dgemv` is more than twice faster on Xeon-Phi we can expect more than $2 \times$ speedup of the reduction on the coprocessor. To verify that, we implemented the reduction to Hessenberg algorithm using Elementary transformation on the Xeon-Phi coprocessor. The code on high level is the same, using the MAGMA MIC APIs [10] to offload the computation to the Xeon Phi. Only certain kernels, like the swapping, had to be specifically designed and optimized. For the other kernels we used MKL, which is highly optimized. We depict in Figure 3 the results in Gflop/s that our implementation achieves, its theoretical upper bound, as well as the performance of `dgemv`. Similarly our implementation reaches asymptotically its upper bound. The gap observed for the Xeon Phi is larger than the one observed for CPU. This is due to hierarchical cache effect and to the fact that the cost of some Level 1 operation is considered marginal on CPU while it is more expensive on Xeon Phi introducing this difference. We implemented an optimized `xswap` kernel since we need to swap both row and column at every step. In our parallel swap implementation we had to design our parallelism to force each set of thread to read/write data aligned with cache for optimal performance. This was useful for column swapping since data is coalescent in memory while for row swapping we had to think differently. For the row swapping we only swap the rows within the current panel and delay the remaining till the end of the panel factorization when we swap the remaining rows in parallel. We split the thread pool over the data so that every set of threads tries to read/write as much as possible data within the same bank of memory. Also another improvement we had to implement for the Xeon Phi, it is not always advantageous to use all the threads i.e 240 threads for Level 1 BLAS because of the overhead of OMP thread management as OMP puts active threads in sleep mode after certain period and wakes them up when necessary. Since the swap function is needed for every step (n times) this overhead become unaffordable. To reduce this overhead we have changed the number of threads dynamically based on the number of elements needed to be swapped.

Figure 4 shows the performance of the reduction on CPUs vs. Xeon Phi. As analyzed, the reduction on the Xeon-Phi is about $2.5 \times$ faster than on the CPUs. The Xeon-Phi implementation is native (uses only the Phi) and thus the CPU is idle or can performs other work. We have implemented a hybrid version that uses both CPU and Phi. Since the reduction is bound by the `dgemv` performance and also since both the `dgemv` and the `dgemm` performance achieve higher

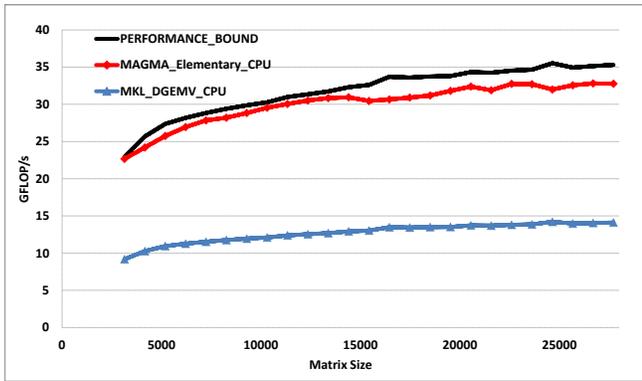


Figure 2. Performance bound for CPU

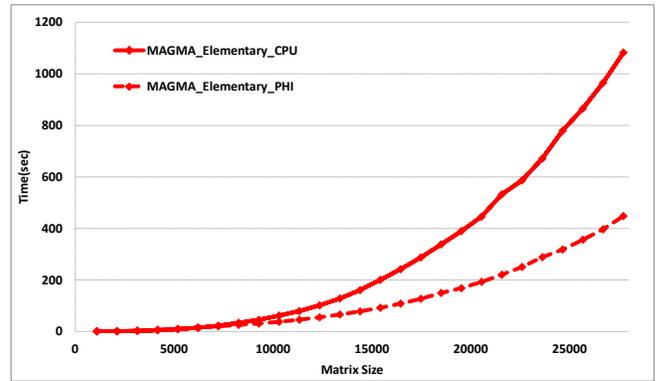


Figure 4. Accelerating the Hessenberg reduction on Xeon Phi

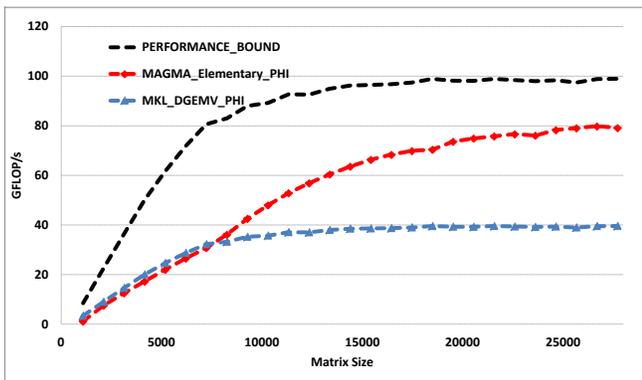


Figure 3. Performance bound for Intel Xeon Phi

ratio on the Phi, the hybrid implementation was around 5% slower and consumed more resources. For the hybrid version the `dgemv` and the `dgemm` are performed on the Phi, otherwise the performance drops down more than 15%. So, only the Level 1 BLAS operations are computed on the CPU since the CPU can handle better Level 1 routines. However, the cost of copying a vector back and forth between the CPU and the Phi at every step is negating the gain obtained and slows down the overall performance by 5%.

EXPERIMENT RESULTS

We performed a set of experiments to compare the performance of our proposed Hessenberg reduction using stabilized elementary transformations with the classical reduction that uses Householder transformations on both CPUs and Xeon-Phi. For the comparison we use the `dgehrd` routine from MKL for CPUs, as well as for the Phi in native mode (using only the Xeon-Phi). We illustrate in Figure 5 the required elapsed time in seconds to perform either the classical `dgehrd` or our `MAGMA_Elementary` on both CPU and Phi. First of all, we should mention that both implementations (`Magma_Elementary` or `MKL_Householder`) are optimized and reach their theoretical upper bounds on either CPUs or Phi. As expected, our proposed Elementary reduction implementation is between 20% to 30% faster than the

Householder one on either the CPU or Phi architecture. According to equations (7) and (8), the Elementary transformations reduce the amount of Level 3 BLAS operations by 62% while keeping the same amount of Level 2 operations. This results in reducing the overall cost of the Hessenberg reduction by 20%. The other optimizations that we have implemented, such as finding the pivot and directly scaling the corresponding vector at the same time as well as the optimized parallel row/column swapping, gave us an additional 5% to 10% improvement. Finally, our proposed Elementary implementation showed fast and efficient reduction to Hessenberg form and was accelerated by more than $2.3\times$ by the use of the Xeon-Phi coprocessors.

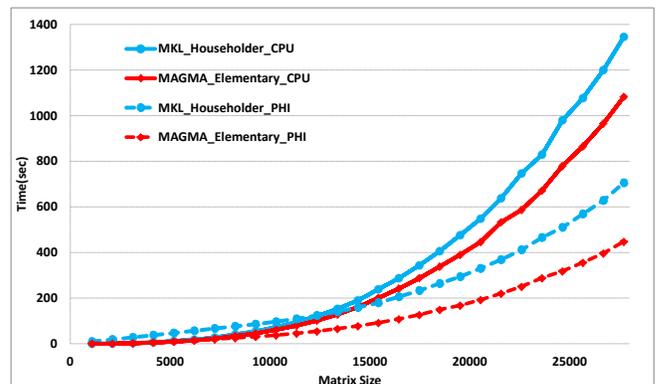


Figure 5. Performance comparison of CPUs vs. Xeon Phi

CONCLUSIONS AND FUTURE WORK

We derived for a first time a blocked algorithm for the reduction to upper Hessenberg form using stabilized elementary transformations and developed highly optimized implementations for multicore CPUs and Xeon Phi coprocessors. The blocking significantly improved the performance of the approach, and even made it 20 to 30% higher than the standard, Householder-based approach. Still, both approaches are memory bound as they feature the same amount of flops in

Level 2 BLAS operations. We designed a model for the theoretical peak for both approaches that clearly shows their limitations, and also illustrates how optimal our implementations are. In particular, we reach up to 95% of the peak on multi-core CPUs and up to about 80% on the Xeon Phi architecture. Our future work will explore the feasibility of using random butterfly transformations to avoid the need for pivoting, while still getting acceptable stability. Further, we will study the reduction of nonsymmetric matrices to band or tridiagonal form using elementary transformation.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. ACI-1339822, the Department of Energy, and Intel. The results were obtained in part with the financial support of the Russian Scientific Fund, Agreement N14-11-00190.

REFERENCES

- Anderson, E., Bai, Z., Bischof, C., Blackford, S. L., Demmel, J. W., Dongarra, J. J., Croz, J. D., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. C. *LAPACK User's Guide*, Third ed. Society for Industrial and Applied Mathematics, Philadelphia, 1999.
- Bischof, C., and van Loan, C. The WY representation for products of Householder matrices. *J. Sci. Stat. Comput.* 8 (1987), 2–13.
- Bischof, C. H., Lang, B., and Sun, X. Algorithm 807: The SBR Toolbox—software for successive band reduction. *ACM Transactions on Mathematical Software* 26, 4 (2000), 602–616.
- Dongarra, J., Gates, M., Haidar, A., Kabir, K., Luszczek, P., Tomov, S., and Yamazaki, I. MAGMA MIC 1.3 Release: Optimizing Linear Algebra for Applications on Intel Xeon Phi Coprocessors. http://icl.cs.utk.edu/projectsfiles/magma/pubs/MAGMA_MIC_SC14.pdf, Nov. 2014.
- Dongarra, J. J., and Moler, C. B. EISPACK: A package for solving matrix eigenvalue problems. 1984, ch. 4, 68–87.
- Dongarra, J. J., Sorensen, D. C., and Hammarling, S. J. Block reduction of matrices to condensed forms for eigenvalue computations. *Journal of Computational and Applied Mathematics* 27, 1-2 (1989), 215 – 227. Special Issue on Parallel Algorithms for Numerical Linear Algebra.
- Francis, F. The QR transformation, part 2. *Computer Journal* 4 (1961), 332–345.
- Geist, G. Reduction of a general matrix to tridiagonal form. *SIAM J. Mat. Anal. Appl* 12 (1991), 362–373.
- Grimes, R. G., and Simon, H. D. Solution of large, dense symmetric generalized eigenvalue problems using secondary storage. *ACM Transactions on Mathematical Software* 14 (September 1988), 241–256.
- Haidar, A., Cao, C., Yarkhan, A., Luszczek, P., Tomov, S., Kabir, K., and Dongarra, J. Unified development for mixed multi-gpu and multi-coprocessor environments using a lightweight runtime environment. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium, IPDPS '14*, IEEE Computer Society (Washington, DC, USA, 2014), 491–500.
- Haidar, A., Tomov, S., Dongarra, J., Solca, R., and Schulthess, T. A novel hybrid CPU-GPU generalized eigensolver for electronic structure calculations based on fine grained memory aware tasks. *International Journal of High Performance Computing Applications* 28, 2 (May 2014), 196–209.
- Howell, G. W., and Diaa, N. Algorithm 841: Bhes: Gaussian reduction to a similar banded hessenberg form. *ACM Trans. Math. Softw.* 31, 1 (Mar. 2005), 166–185.
- Howell, G. W., Geist, G., and Rowan, T. Error analysis of reduction to banded hessenberg form. *Tech. Rep. ORNL/TM-13344*.
- Intel. Math kernel library. <https://software.intel.com/en-us/en-us/intel-mkl/>.
- Kågström, B., Kressner, D., Quintana-Orti, E., and Quintana-Orti, G. Blocked Algorithms for the Reduction to Hessenberg-Triangular Form Revisited. *BIT Numerical Mathematics* 48 (2008), 563–584.
- Karlsson, L., and Kågström, B. Parallel two-stage reduction to Hessenberg form using dynamic scheduling on shared-memory architectures. *Parallel Computing* (2011). DOI:10.1016/j.parco.2011.05.001.
- Lang, B. Efficient eigenvalue and singular value computations on shared memory machines. *Parallel Computing* 25, 7 (1999), 845–860.
- Martin, R., and Wilkinson, J. Similarity reduction of a general matrix to Hessenberg form. *Numerische Mathematik* 12, 5 (1968), 349–368.
- Rutishauser, H. Solution of eigenvalue problems with the LR transformation. *Nat. Bur. Standards Appl. Math. Ser.* 49 (1958), 47–81.
- Schreiber, R., and van Loan, C. A storage-efficient WY representation for products of Householder transformations. *J. Sci. Stat. Comput.* 10 (1991), 53–57.
- Tomov, S., Nath, R., and Dongarra, J. Accelerating the reduction to upper Hessenberg, tridiagonal, and bidiagonal forms through hybrid GPU-based computing. *Parallel Comput.* 36, 12 (Dec. 2010), 645–654.
- Van Loan, C. Using the hessenberg decomposition in control theory. 102–111.
- Wachspress, E. L. similarity matrix reduction to banded form. *manuscript* (1995).
- Wilkinson, J. H. Householder's method for the solution of the algebraic eigenproblem. *The Computer Journal* 3, 1 (1960), 23–27.

Efficient Algorithms for Improving the Performance of Read Operations in Distributed File System

**T Lakshmi Siva Rama
Krishna**
Jawaharlal Nehru Institute of
Advanced Studies (JNIAS)
Hyderabad, INDIA
sivamca.mtech@gmail.com

T Ragnathan
Computer Science and
Engineering
ACE Engineering College
Hyderabad, INDIA
ragu_savi@yahoo.com

Sudheer Kumar Battula
Computer Science and
Engineering
ACE Engineering College
Hyderabad, India
sudheer.itdict@gmail.com

ABSTRACT

Distributed file systems (DFSs) are used in the modern cloud-based systems to store and process a large amount of data. File sharing semantics is used by the DFS for sharing the data in a consistent manner among the authorized users of the system. The limitation of popular session semantics is that read client processes cannot read the modifications done by a concurrent write client process on the same shared file in the same session. Linearizability semantics followed in the BlobSeer DFS permits the read client processes to read the previous version of a binary large object (blob) while write operation is carried out on that blob concurrently. In this paper, we have proposed a new type of semantics and metric namely "Speculative Semantics" and "Currency" respectively. Speculative semantics permits the read client processes to read the modifications done by the concurrent write client process in the same session. Currency is the metric used for measuring the performance of the read algorithms. In this paper, we have proposed two new read algorithms based on speculative semantics. We have conducted experiments on Blobseer DFS to measure the performance of these read algorithms and the results obtained indicate that the proposed algorithms perform better than existing read algorithm of the BlobSeer DFS.

Author Keywords

Distributed System; BlobSeer; Concurrency; Speculation; Performance;

ACM Classification Keywords

D.4.3 File Systems Management: Distributed File System

INTRODUCTION

Many IT organizations use distributed file systems (DFSs) to store, process and access a large amount of data generated from various sources. File sharing semantics used by the file systems describes how multiple user programs can access a shared file simultaneously.

Most of the DFSs use session semantics for file sharing purpose. The limitation of this semantics is that read client processes cannot read the modifications done by a concurrent write client process on the same shared file in the same session. However, the read client process can read the modified contents from the shared file during the next session initiated by that read client process. BlobSeer DFS [7] follows linearizability semantics which permits to have multiple versions for a binary large object (blob). Linearizability semantics permits the read client processes to read the previous version of the blob while concurrent write operation is carried out on that blob for creating a new version for that blob.

In this paper, we have proposed a new type of semantics and metric namely "Speculative Semantics" and "Currency" respectively. Speculative semantics permits the read client processes to read the modifications done by the concurrent write client process in the same session. Currency is a metric which is useful to know whether the data read from the file is recently modified or not. In this paper, we have also proposed two read algorithms for the DFS based on speculative semantics. We have measured the performance of read algorithms based on the metric "Currency". The performance results indicate that the proposed algorithms perform better than the algorithm followed in the BlobSeer DFS.

The body of the paper is organized as follows. In the next section, we describe related work. In Section 3, we explain data consistency, the proposed speculative semantics, data currency and example applications. In Section 4, we describe the proposed read algorithms of the DFS. In Section 5, we present performance results. Section 6 covers the conclusion and future works.

RELATED WORK

Four types of file sharing semantics have been proposed in the literature [12]. They are unix semantics, session semantics, immutable semantics and transactional semantics.

According to unix semantics, modifications performed on a shared file F_0 at time instance t_1 must be available to all other read operations on F_0 at time instance t_2 , if $t_2 > t_1$. Implementing unix semantics in a distributed system (DS) is difficult due to its communication overhead and maintenance of consistent replicas. Session semantics is used by the network file system (NFS)[9] and Andrew file system (AFS)[4]. A *file session* consists of multiple system calls starting with

the *open* system call and ending with the corresponding *close* system call. All other operations like read and write are performed between the *open* - *close* pair. Under session semantics, a read client process can read the modified content of a shared file only after the close operation performed by the corresponding concurrent write client process on that file.

Under immutable semantics modifications cannot be performed on existing file and hence it greatly simplifies the sharing and replication of files in the DFS. Two important properties of immutable file are: (i) Once a file is created, its contents cannot be modified (ii) File name cannot be reused [6].

The coda file system (CFS) [10] used transactional semantics for file sharing purpose. According to transactional semantics, a file session or a set of file sessions should be atomic and atomicity ensures that either all of the operations in a file session will be performed or none of the operations will be performed. A client process executes a primitive *begin_transaction* to signal that what follows must be executed indivisibly. After this, system calls like *read* and *write* can be executed for performing read and write operations on one or more files of a DFS. A primitive *end_transaction* is executed after completion of the the requested work. Under transactional semantics, a read client process cannot read the modifications done by a concurrent write client process, if that write client process has not completed its file session by executing the primitive *end_transaction*.

Google’s proprietary file system is called as Google File System (GFS) which used a relaxed consistency model [2] for file sharing purpose, where as HDFS [11] is the open source implementation of GFS, which supports single writer multiple reader model. Blobseer DFS [7] adopts linearizability semantics [3] for file sharing purpose. In, Blobseer DFS, data is abstracted as a long sequence of bytes called as binary large objects (blobs) and multiple versions are available for each blob, that is for each upate on a blob, a new version is created. Under linearizability semantics, while modifications are carried out on a shared blob by a concurrent write client process (*WCP*), a concurrent read client process (*RCP*) will be able to read the previous version of that blob. In other words, the *RCP* can read only the currently available version of the blob.

In the literature, speculation based protocols are proposed for database systems to improve the performance of transaction processing [5][8]. In this paper, we have defined a new type of semantics namely ”Speculative Semantics” and a measure for evaluating the performance of the read algorithms namely ”Currency”. We have proposed two read algorithms for a DFS based on speculative semantics. Our algorithms permit the read client processes to read the modified data while update operations are carried out on the same data by a concurrent write client process provided such a reading does not affect data consistency.

DATA CONSISTENCY, SPECULATIVE SEMANTICS, DATA CURRENCY AND EXAMPLE APPLICATIONS

In this section, we discuss regarding data consistency, the proposed speculative semantics, data currency and example applications.

Data Consistency

Let us consider that in a DFS, two or more processes are trying to read the same data. As per [2], the data read by the processes is consistent “if all the processes will be able to access the same data regardless of the site which they read from”. The author of [13] has discussed two types of data consistencies namely strong consistency and weak consistency. Let us consider a scenario where one *WCP* is modifying a shared data object *d* to *d'* and concurrently an *RCP* is also trying to read the same *d*. If that *RCP* is completed after the completion of *WCP* then *RCP* should have read the modified data object *d'* if the DFS supports strong data consistency [13]. If the DFS supports weak consistency then *RCP* may or may not read *d'* [13]. Strong data consistency is very important for real time applications. In this paper, we have considered strong data consistency.

Speculative semantics

Speculative semantics permits writes to an open file to be visible to other concurrent client processes which have opened the same file, provided such a reading does not affect the data consistency. Thus speculative semantics provides strong consistency. We explain the advantage of speculative semantics through an example. Let us consider, a file F1 is created at time instance 5 and a *WCP* starts modification on F1 at time instance 10. The *WCP* produced a new version of F2 of F1 at time instance 40 and also produced new version F3 of F2 at time instance 150. Now assume, *RCP* starts its execution at time instance 50 for reading F1. This scenario is depicted in Figure 1. If session semantics is followed then *RCP* reads F1 and if linearizability semantics is followed then *RCP* reads F2. Note that, if speculative semantics is followed then *RCP* will be able to read F3 which is the most recent version available in the DFS.

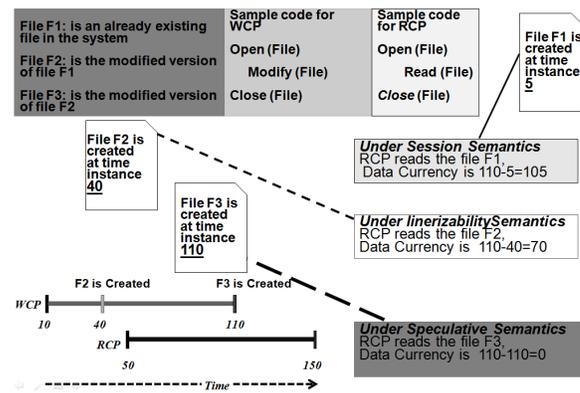


Figure 1. Comparison of Session, Linearizability and Speculative Semantics

Under session semantics and linearizability semantics an *RCP* can only read current version of a file and it cannot read the new version of that file which is currently being created

in the DFS. Where as under speculative semantics, an *RCP* can read the new version of a file which is currently being created in the DFS, provided such a reading does not affect data consistency.

Currency

Currency is a metric, which is useful to know whether the data read from a file is the most recently created one or not. We have defined the currency based on [1]. The currency value of a data object computed for a read client process is the difference between the creation time of the data object read by the read client process from the creation time of the most recent version of that data object which is created before the completion of read client process. Note that, currency value will always be positive and high positive value indicates low data currency, low positive value indicates high data currency and a zero indicates highest data currency. Thus, if the computed currency value zero indicates that a read client process has read the most recent version of a file.

We explain the computation of currency under session, linearizability and speculative semantics for the same scenario depicted in Figure 1. We have shown in Figure 1 that an *RCP* has started its execution at time instance 50 to read F1 and completed its execution at time instance 150. Under session semantics, the *RCP* reads F1 and hence the computed currency value is 105 (110-5). Under linearizability semantics, the *RCP* reads F2 and the currency value computed is (70) 110-40. Where as under speculative semantics, the *RCP* reads F3 and the currency value computed is 0 (110-100) which indicates that the *RCP* has read the most recent version of the file. Note that, under session and linearizability semantics, an *RCP* can read only previous version of a file even though new version of that file is available at the time of its completion. Whereas under speculative semantics, an *RCP* could have read a version of the file which is available at the time of its completion. Hence, under speculative semantics an *RCP* can read the most recent version of a file created in the DFS.

Example Applications

The proposed speculative semantics is very useful for applications like stock trading and real time systems to read the recently modified data from the file system.

Consider a stock trading application which receives a market feed of stock price changes occurring during the trading day and consider that these stock prices are stored in DFS. Let us assume that a particular stock price (S1) is stored in file F4 and updated at time t3. Let us also assume that a client (C1) of stock trading application reads S1 at time t4 and $t4 > t3$. If session semantics is followed by the DFS then C1 may read the updated value of S1 or not depends upon the time of closing the session. If linearizability semantics is followed by the DFS then C1 may read the updated value of S1 or not depends upon the time of completion of the *write* system call which has modified S1. If speculative semantics is followed then C1 can read the updated value of S1 as $t4 > t3$. So, C1 can take a better decision if speculative semantics is followed by the DFS.

In real time system like flight control system, the data read by various sensors placed in the plane can be stored in the file system which is entirely maintained in the main memory. In such a system, speculative semantics is useful for the flight control applications to read the most recently modified data from the file system so that correct decisions can be taken with regard to the system operation.

SPECULATIVE SEMANTICS-BASED ALGORITHMS

In this section, first, we describe the architectural components of a DFS. Next, we discuss the notations and assumptions used in the existing and the proposed algorithms. Then, we describe the existing read algorithm of a DFS, explain our proposed speculative semantic-based read (SSR) algorithm and its limitation. Finally, we describe the proposed improved speculative semantics based read (ISSR) algorithm for the DFS.

DFS Architecture

The architectural elements of a DFS are master node (MN) and data nodes (DNs). The MN stores meta data and DN stores the files. Note that, DNs are also used to execute client processes (user application programs). In each DN, a DFS client program is getting executed. Note that, communication with MN and DNs and accessing data stored in the DN are done using this program. We have assumed that the DFS supports client-side caching technique and the caches are maintained in the DNs of the DFS. Caches will be affected during file read and file write operations carried out on the DFS.

Notations and Assumptions

Notations: The notations used in the proposed algorithm and their meanings are described below.

- *RCP*: read client process
- *WCP*: write client process
- *F5, F6*: files accessed by *RCPs* and *WCPs*
- *Read* : read procedure for DFS
- *Write*: write procedure for DFS
- *MT*: main thread of *Read*
- *ST*: speculative thread of *Read*
- *SBO*: starting byte offset
- *EBO*: ending byte offset
- *Readbuf*: buffer used by the *MT* of *Read* for storing data read from *F5*
- *Writebuf*: buffer used to store data modified by the *Write* for *F5*
- *Stbuf*: buffer used to store data read by the *ST* of *Read* from the *SM* of data node where *WCP* is getting executed.
- *Tempbuf*: buffer used to store temporary data read by the *MT* of *Master_Read(F5,SBO,EBO,Readbuf)* from the data nodes where the un-modified blocks are available.

- *ubs*: blocks updated by *WCP*
- *SM*: shared memory

Assumptions:

- A master node (MN) is the node where the metadata of the DFS is stored and data node (DN) is the node where the files are stored. Only one MN and one or more DNs are available in the distributed file system.
 - Multiple *RCPs* and *WCPs* are concurrently accessing the file (*F5*) and at least one block is common in the range of blocks accessed by *RCPs* and *WCPs*. *RCPs* and *WCPs* can run in same or different DN.
 - A *WCP* executes the *write* system call only once in a session.
 - Whenever a file is getting modified that status is updated in the MN
 - *Existing_Read(F5,SBO,EBO)* is the already existing procedure specific to the DFS which is used for reading the range of bytes from *F5*
 - *Master_Read(F5,SBO,EBO,Readbuf)* is the *MT* executed by the *Main Read Procedure (F5,SBO,EBO,Readbuf)* to read the specified blocks from the DNs
 - *Speculative_Read(F5,SBO,EBO,Stbuf)* is the *ST* executed by the *Main Read Procedure (F5,SBO,EBO,Readbuf)* to read the updated blocks from the main memory buffers where *WCP* is getting executed.
 - *Check_MN(F5)* is the function used to verify whether *F5* is currently getting modified or not, by checking with MN. If *F5* is modified, this function returns “*true*” else the function return value is “*false*”
 - Both *ST* and *MT* are executed concurrently
 - New version for *F5* is *F6*
 - *Read_Metadata(F5,SBO,EBO)* is the procedure used to obtain a list of data blocks and nearest DNs where these data blocks are stored.
 - The *Write* procedure registers with MN by providing the following details: (i) File name(ii) IP address of the system where *WCP* is getting executed (iii) *SBO* and *EBO* of the data to be modified in the file.
- Next, we discuss the existing read algorithm of DFS.

Existing Read Algorithm

In the existing read algorithm of a DFS, to read a specified range of bytes from a file, first DFS client requests MN. In response to this, MN verifies whether the file is available in the file system or not. If the file is not available an error message is generated. If the file is available, then MN sends the meta data of that file to the requested DFS client. Then the DFS client fetches the required file blocks from the nearest data nodes in a parallel manner.

Algorithm: Existing_Read(F5,SBO,EBO,Readbuf)

```

1: if F5 is not available then
2:   Send File not available error message to the client
3: else
4:   Read_Metadata (F5, SBO, EBO)
5:   for all b blocks of F5 in parallel do
6:     read b from corresponding nearest DN into Readbuf
7:   end for
8: end if

```

The limitation of existing read algorithm of a DFS is that, an *RCP* can read only previous version of a file even though new version of that file is available by the time that *RCP* completes its execution. Hence, we have proposed SSR and ISSR algorithms and modified the existing Blobseer DFS to evaluate the performance of our proposed algorithms

Proposed SSR Algorithm

To read a range of bytes from a file, the DFS client contacts MN by providing name of the file, range of bytes to be read from the file and address of the local buffer where the data will be stored. In response to this, first, MN verifies whether the file is available in the file system or not. If the file is not available an error message is generated. If the file is available, MN calculates which blocks of the file consists the requested range of bytes (range of bytes in a file are the address locations of the file to be read or modified). Meanwhile the DFS client requests the MN to check whether the requested file is getting modified or not. If file is not getting modified, existing read procedure of the DFS is followed. If the file is getting modified, then *ST* is created and the *ST* starts reading the modified blocks from the main memory buffers of the system where *WCP* is getting executed. Meanwhile, MN sends the list of nearest DNs, which have the replicated copies of the requested file blocks to the DFS client. After this the *MT* fetches the first file block (a file block is the subset of a file) from the nearest DN and places in its local buffer. Note that, both *ST* and *MT* are executed in a parallel manner. If multiple blocks have to be read by *MT*, then fetching of the file blocks from various DNs can be performed in a parallel manner. Next, whether new version of the file is created or not will be verified with MN. If the new version of the file is available then we combine the contents read by *ST* and *MT*. Otherwise, *RCP* will read previous version of the file without affecting data consistency.

The advantage of this algorithm is that read client processes are able to read the modified consistent data while updates are carried out by another write client process on the same file.

Algorithm: Main Read Procedure (F5,SBO,EBO,Readbuf)

```

/* This is the Main Read procedure */
1: if F5 is not available then
2:   Send File not available error message to the client
3: else
4:   Value=Check_MN(F5)
5:   if Value="true" then
6:     Execute Speculative_Read(F5,SBO,EBO,Stbuf)

```

```

/* This is ST*/
7: Execute Master_Read(F5,SBO,EBO,Readbuf)
/*This is MT*/
8: /* Both MT and ST are executed in parallel */
9: else
10: Execute Master_Read(F5,SBO,EBO,Readbuf)
/*This is MT*/
11: end if
12: end if
Speculative_Read(F5,SBO,EBO,Stbuf)
1: ST is created
2: ST gets the IP address of the system where WCP is getting executed.
3: for all ubS of F5 in parallel do
4: read the updated block from WCP and stores that block into Stbuf
5: end for
6: Check with MN to know whether F6 is created or not
7: /* F6 is the new version of F5 */
8: if F6 is available then
9: wait for the completion of MT
10: if MT is completed then
11: Combine the Readbuf and Stbuf contents into Readbuf
12: Terminate MT
13: end if
14: else
15: Wait for the message from MT
16: end if
Master_Read(F5,SBO,EBO,Readbuf)
1: Read_Metadata (F5, SBO, EBO)
2: for all b blocks of F5 do
3: read the block from the nearest DN into Readbuf
4: end for
5: Check with MN to know whether F6 is created or not
6: /* F6 is the new version of F5 */
7: if F6 is available then
8: wait for the message from ST
9: else
10: Terminate ST
11: end if

```

The limitation of this SSR algorithm is that the read access time will be slightly more than that of the existing read algorithm of BlobSeer DFS. Hence, we have proposed an improved algorithm namely ISSR for reducing the read access time. We explain the advantage of our ISSR algorithm over SSR with an example which is depicted in Figure 2. Let us assume that a file F7 consists of 100 pages and a WCP is modifying the pages 40 to 79. If we use SSR algorithm in this scenario, then MT will read all the 100 pages and ST will read only the modified pages 40 to 79. For the same scenario, if we use ISSR algorithm, then MT will have to read 1 to 39 and 80 to 100 pages (only unmodified pages) from the hard disk of the data nodes and ST will read only modified pages 40 to 79 from main memory buffers maintained by WCP. Hence, in ISSR, less number of blocks are read from the hard disk than that of SSR. Hence, ISSR algorithm can perform better than SSR algorithm. The flowchart for ISSR

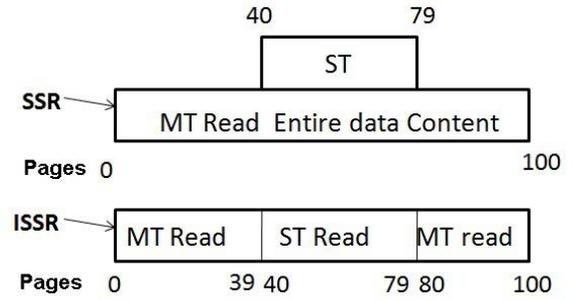


Figure 2. SSR vs ISSR

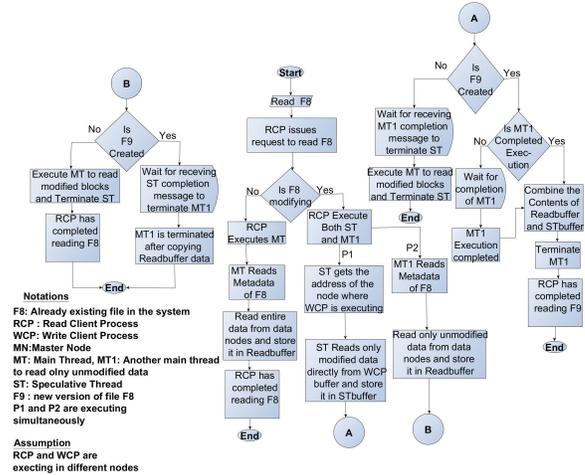


Figure 3. ISSR Flowchart

algorithm is depicted in Figure 3. In SSR algorithm, whenever a client requests for a file, the MT starts reading both the unmodified and modified pages (page is the smallest subset of a file) from the DNs, whereas in the ISSR algorithm MT reads only the unmodified pages from the DNs and ST reads only the modified pages from the main memory of WCP. In the flowchart we have used notations MT and MT1 (main thread 1). MT is used for the cases where the shared file is not modified by a concurrent WCP and MT1 is used for the cases where file is getting modified by a concurrent WCP. Finally, both the modified and unmodified pages are combined to produce the requested pages of the RCP. The advantage of ISSR algorithm over SSR algorithm is that the read access time is reduced without compromising on the data currency.

Discussion

The SSR algorithm makes use of existing read procedure of the DFS for reading both modified and unmodified blocks from the data nodes. The ISSR algorithm uses existing read procedure of the DFS for reading only unmodified blocks. If any client-side caching technique is supported in the existing read procedure of the DFS, then that will automatically be used by both SSR and ISSR algorithms. The proposed SSR and ISSR algorithms will read the modified blocks from the main memory buffer of the WCP (address space of WCP) and hence these algorithms will not disturb the caching policy of existing write procedure of the DFS.

PERFORMANCE RESULTS

In this section, first we discuss regarding architectural components of Blobseer DFS, next we explain our test platform and then we discuss the overview of experiments conducted. Finally, we present the experimental results.

Architectural Components of BlobSeer DFS

Data is abstracted in BlobSeer DFS as long sequences of bytes called blobs (binary large objects). Note that, a blob consists of one or more pages. A read procedure can read only one blob at a time. The read operations are carried in an atomic manner by Blobseer DFS that is either all the required pages of the blob are read or nothing is read. Multiple versions (snapshots) of blobs are available in the BlobSeer DFS to support concurrent access.

The architectural components of BlobSeer DFS are Clients, Data Providers or simply Providers, Provider Manager, Metadata Providers and Version Manager. Let us discuss regarding these components one by one.

- *Clients* can create, read, write and append data from/to blobs in a concurrent manner.
- *Data Providers* physically store the pages generated by append operations and write operations.
- *The Provider Manager (PM)* keeps information regarding the available storage space and schedules the placement of newly generated pages.
- *Metadata Providers (MP)* physically stores the metadata that allows identifying the pages that make up a snapshot version.
- *Version Manager (VM)* is in charge of assigning new snapshot version numbers to blobs.

Test Platform

Our test platform is built on a cluster consisting of eight nodes. Each node has an Intel Core 2 Duo Processor of 2.9 GHz, 2GB of RAM, 500GB SATA HDD. We have installed Ubuntu-12.04 with kernel 3.2.1 in these nodes and on top of Ubuntu the Blobseer version 1.1 is installed. In these eight nodes, meta data provider is installed in one node and version manager and provider manager are installed in two different nodes. Remaining six nodes are used as data providers. We have modified the read procedure of BlobSeer DFS as per our proposed SSR and ISSR algorithms.

Overview of Experiments

The experiments are conducted for two scenarios and the page size of 32KB is fixed for both the scenarios. We have considered that *RCPs* read entire blob which consists of 1000 pages.

Scenario 1: We have varied the number of *RCPs* from 8, 16, 24, 32 and 40 and *WCPs* from 2, 4, 6, 8 and 10 executed in the cluster and have fixed the number of pages getting modified is equivalent to 25. Then we have calculated the average data currency and average read access time for the existing linearizability semantics-based read algorithm (LSR) of Blobseer DFS, proposed SSR and ISSR.

Scenario 2: We have varied the number of pages getting modified from 10,20,30,40 and 50. Also, we have fixed the number of *RCPs* to 16 and *WCPs* to 4 in the cluster. Then, we have calculated the average data currency and average read access time for the algorithms LSR, SSR and ISSR.

Experimental results

Figure 4 and Figure 5 depicts the average data currency performance of *RCPs*. Figure 4 shows the average data currency performance by varying the number of processes (*RCPs* from 8, 16, 24, 32 and 40 and *WCPs* from 2, 4, 6, 8 and 10) executed in the cluster and Figure 5 shows the average data currency performance by varying the number of pages (10,20,30,40 and 50) getting modified in the cluster. We can observe that the average data currency of proposed ISSR and SSR are better than LSR.

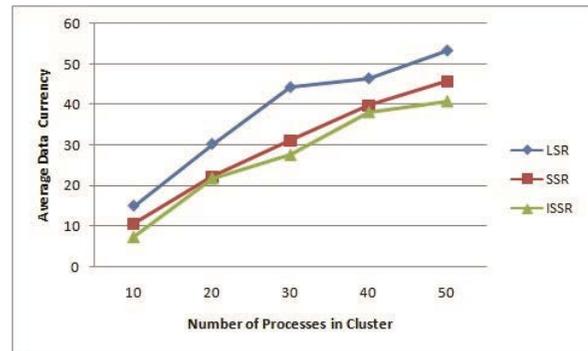


Figure 4. Average data currency versus number of processes vary in the cluster.

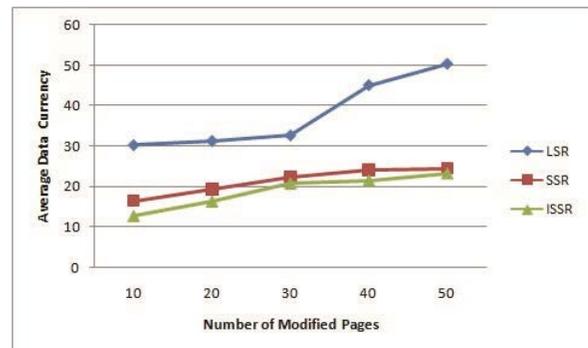


Figure 5. Average data currency time versus number of modified pages in the cluster.

Figure 6 and Figure 7 show the average read access time of *RCPs*. Figure 6 shows the average read access time performance by varying the number of processes (*RCPs* from 8, 16, 24, 32 and 40 and *WCPs* from 2, 4, 6, 8 and 10) executed in the cluster and Figure 7 shows the average read access time performance by varying the number of pages (10,20,30,40 and 50) getting modified in the cluster. We can observe that the average read access time per page in ISSR algorithm is less than the average read access time per page computed for the SSR algorithm. We can also observe that, the average read access time of ISSR algorithm and existing LSR algorithm of BlobSeer DFS are almost same. The advantage of

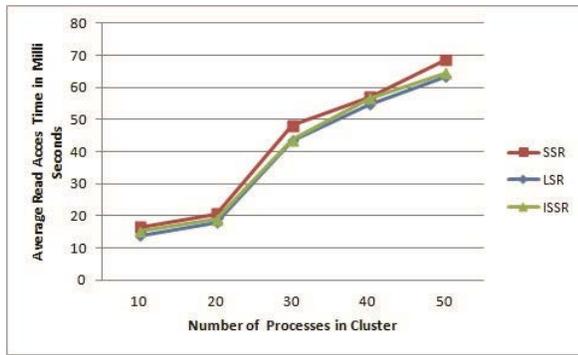


Figure 6. Average read access time versus number of processes vary in the cluster.

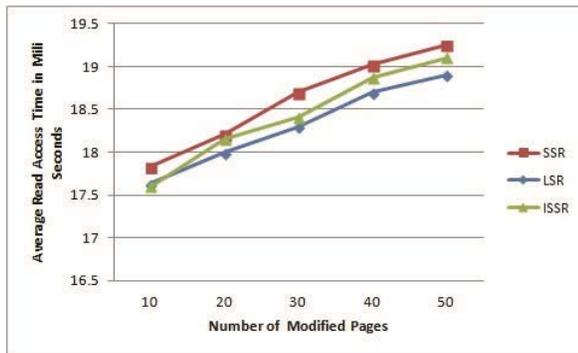


Figure 7. Average read access time versus number of modified pages in the cluster.

our proposed ISSR algorithm is that a read client process is able to read the new version of a blob while modification is done on that blob concurrently. Whereas, existing LSR allows the read client process to read only the previous version of the blob while concurrent modification is done on that blob. Overall, we conclude that the proposed ISSR algorithm performs better than SSR algorithm and LSR algorithm of the BlobSeer DFS by considering the metric “Currency”.

CONCLUSION AND FUTURE WORKS

In this paper, we have proposed a new type of file sharing semantics namely speculative semantics and a new currency metric for the distributed file system. We have also proposed two algorithms based on speculative semantics. These algorithms allow the read client processes to read the most recent version of the file created in the distributed file system without affecting data consistency while modification on that file is carried out concurrently by a write client process. However, these algorithms require that the read client process has to start its execution after that of the concurrent write client process. We have implemented the proposed algorithms in Blobseer distributed file system and conducted experiments. The results of the experiments indicate that the proposed algorithms perform better than the algorithm followed in the BlobSeer distributed file system.

As a part of future work, we wish to propose new algorithms for the cases where long read client processes will be able to read updates performed by the short write client process even though the write client process has started its execution after that of the long read client processes.

REFERENCES

1. Batini, C., Cappiello, C., Francalanci, C., and Maurino, A. Methodologies for data quality assessment and improvement. *ACM Computing Surveys (CSUR)* 41, 3 (2009), 16.
2. Ghemawat, S., Gobioff, H., and Leung, S.-T. The google file system. In *ACM SIGOPS Operating Systems Review*, vol. 37, ACM (2003), 29–43.
3. Herlihy, M. P., and Wing, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990), 463–492.
4. Howard, J. H., Kazar, M. L., Menees, S. G., Nichols, D. A., Satyanarayanan, M., Sidebotham, R. N., and West, M. J. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS)* 6, 1 (1988), 51–81.
5. Krishna Reddy, P., and Kitsuregawa, M. Speculative locking protocols to improve performance for distributed database systems. *Knowledge and Data Engineering, IEEE Transactions on* 16, 2 (2004), 154–169.
6. Levy, E., and Silberschatz, A. Distributed file systems: Concepts and examples. *ACM Comput. Surv.* 22, 4 (Dec. 1990), 321–374.
7. Nicolae, B., Moise, D., Antoniu, G., Bougé, L., and Dorier, M. Blobseer: Bringing high throughput under heavy concurrency to hadoop map-reduce applications. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, IEEE (2010), 1–11.
8. Rangunathan, T., and Krishna Reddy, P. Speculation-based protocols for improving the performance of read-only transactions. *International Journal of Computational Science and Engineering* 5, 3 (2010), 226–242.
9. Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., and Lyon, B. Design and implementation of the sun network filesystem. In *Proceedings of the Summer USENIX conference* (1985), 119–130.
10. Satyanarayanan, M., Kistler, J. J., Kumar, P., Okasaki, M. E., Siegel, E. H., and Steere, D. C. Coda: A highly available file system for a distributed workstation environment. *Computers, IEEE Transactions on* 39, 4 (1990), 447–459.
11. Shvachko, K., Kuang, H., Radia, S., and Chansler, R. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, IEEE (2010), 1–10.
12. Tanenbaum, A. S., and Steen, M. v. *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.
13. Vogels, W. Eventually consistent. *Queue* 6, 6 (2008), 14–19.

Long-time Simulation of Calcium Induced Calcium Release in a Heart Cell using the Finite Element Method on a Hybrid CPU/GPU Node

Xuan Huang

Department of Mathematics and Statistics,
University of Maryland, Baltimore County
1000 Hilltop Circle, Baltimore, MD 21250
hu6@umbc.edu

Matthias K. Gobbert

Department of Mathematics and Statistics,
University of Maryland, Baltimore County
1000 Hilltop Circle, Baltimore, MD 21250
gobbert@umbc.edu

ABSTRACT

A mathematical model of Calcium Induced Calcium Release in a heart cell has been developed that consists of three coupled non-linear advection-diffusion-reaction equations. A program in C with MPI based on matrix-free Newton-Krylov method gives very good scalability, but still requires large run times for fine meshes. A programming model with CUDA and MPI that utilizes GPUs on a hybrid node can significantly reduce the wall clock time. This paper reports initial results that demonstrate speedup using a hybrid node with two GPUs over the best results on a CPU node.

Author Keywords

Calcium Induced Calcium Release, finite element method, GPU, MPI

ACM Classification Keywords

I.6.1 SIMULATION AND MODELING (e.g. Model Development). : Performance

INTRODUCTION

A mathematical model for the calcium induced calcium release (CICR) in a heart cell has been developed [5–7] that consists of three coupled non-linear reaction-diffusion equations, which describe the concentrations of calcium ions (C), fluorescent calcium indicator (F), and the endogenous calcium buffers (B). In [2] (see also [4] for details) a matrix-free Newton-Krylov method for the simulation of calcium induced calcium release in a heart cell was presented. The underlying model of calcium flow is given by a system of three coupled diffusion-reaction equations, in which the occurring source terms are highly nonlinear point sources modeled by Dirac delta distributions. The method is based on a finite element discretization and implemented in a matrix-free manner. The convergence of the finite element method in presence of measure valued source terms, as they occur in the calcium model, was rigorously shown in [11] and numerical results agree well with the theoretical predictions.

This paper extends above by implementing finite element with Newton-Krylov methods on GPUs (graphics processing units). In [2], performance studies show good

scalability, programmed in C with MPI. However, the run time is still huge for large mesh size, requiring excessive time if use few number of nodes. During recent years, general purpose GPUs offer an opportunity to greatly increase the throughput, and implement new ideas in parallel computing.

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model by NVIDIA. The CUDA architecture included a unified shader pipeline, allowing each and every arithmetic logic unit (ALU) on the chip to be marshaled by a program intending to perform general-purpose computations. A typical Tesla K20 GPU that we use from NVIDIA has 2496 cores, and is capable to have a theoretical double precision floating point performance of 1.17 TFLOP/s. The problem discussed in this paper is both computationally intensive and massively parallel, offering a good opportunity for better performance on hybrid nodes with CPUs and GPUs. A problem of reaction-diffusion type is solved in [8], where the authors compare different time-integration methods with a 2D model using one GPU. A Jacobian-free Newton-Krylov method with GPU acceleration is discussed in [9], where the problem size is limited by using one GPU. Our method solves a PDE system on three-dimensional domain with a degrees of freedom more than 25 million. We use performance on one state-of-the-art 16-core CPU node as baseline for speedup computation, rather than serial CPU performance on one core. Our implementation with MPI and CUDA is scalable to multiple nodes with multiple GPUs. This paper reports initial results on one node with 2 GPUs available.

BACKGROUND

The three-species application problem models the flow of calcium on the scale of one heart cell. Calcium ions enter into the cell at calcium release units (CRUs) distributed throughout the cell and then diffuse. At each CRU, the probability for calcium to be released increases along with the concentration of calcium, thus creating a feedback loop of waves re-generating themselves repeatedly. An accurate model of such waves is useful since they are part of the normal functioning of the heart, but can also trigger abnormal arrhythmias. This model requires simulations on the time scale of several repeated

waves and on the spatial scale of the entire cell. This requires long-time studies on spatial meshes that need to have a high resolution to resolve the positions of the calcium release units throughout the entire cell [2].

The problem can be modeled by a system of coupled, non-linear, time-dependent advection-diffusion-reaction equations

$$u_t^{(i)} - \nabla \cdot (D^{(i)} \nabla u^{(i)}) + \beta^{(i)} \cdot (\nabla u^{(i)}) = q^{(i)} \quad (1)$$

of $i = 1, \dots, n_s$ species with $u^{(i)} = u^{(i)}(\mathbf{x}, t)$ representing functions of space $\mathbf{x} \in \Omega \subset \mathbb{R}^3$ and time $0 \leq t \leq t_{\text{fin}}$. The diffusivity matrix $D^{(i)} = \text{diag}(D_{11}^{(i)}, D_{22}^{(i)}, D_{33}^{(i)}) \in \mathbb{R}^{3 \times 3}$ consists of positive diagonal entries, which are assumed to dominate the scale of the advection velocity vectors $\beta^{(i)} \in \mathbb{R}^3$, so that the system is always of parabolic type. We consider the rectangular domain shown in Figure 1, where numerical mesh is also demonstrated in a very coarse way. The model also incorporates no-flux bound-

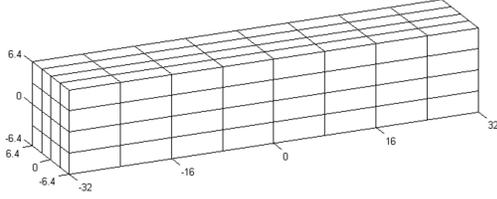


Figure 1. Rectangular domain, units in μm .

ary conditions

$$\mathbf{n} \cdot (D_i(\mathbf{x}) \nabla u^{(i)}) = 0 \quad \text{for } x \in \partial\Omega, 0 < t \leq t_{\text{fin}} \quad (2)$$

and a given set of initial conditions

$$u^{(i)}(\mathbf{x}, 0) = u_{\text{ini}}^{(i)}(\mathbf{x}) \quad \text{for } x \in \Omega, t = 0. \quad (3)$$

The right-hand side $q^{(i)}$ of each PDE in (1) is written in a way that distinguishes the different dependencies and effects as

$$q^{(i)} = r^{(i)}(u^{(1)}, \dots, u^{(n_s)}) + s^{(i)}(u^{(i)}, \mathbf{x}, t) + f^{(i)}(\mathbf{x}, t). \quad (4)$$

Our consideration of this problem is inspired by the need to simulate calcium waves in a heart cell. The general system (1) consists of $n_s = 3$ equations corresponding to calcium ($i = 1$), an endogenous calcium buffer ($i = 2$), and a fluorescent indicator dye ($i = 3$). We describe terms on the right hand side (4) as following: The term $f^{(i)} = f^{(i)}(\mathbf{x}, t)$ incorporates a scalar linear test problem that is been discussed in [3], here we set $f^{(i)} \equiv 0$ for all i . Terms belonging only to the first equation are multiplied with the Kronecker delta function δ_{i1} , where $i = 1, \dots, n_s$,

$$s^{(i)}(u^{(i)}, \mathbf{x}, t) = (-J_{\text{pump}}(u^{(1)}) + J_{\text{leak}} + J_{SR}(u^{(1)}, \mathbf{x}, t)) \delta_{i1}. \quad (5)$$

These are the nonlinear drain term J_{pump} , the constant balance term J_{leak} and the key term of the model J_{SR} .

This term houses the stochastic aspect of the model, since the calcium release units (CRUs) which are arranged discretely on a three-dimensional lattice, each have a probability of opening depending on the concentration of calcium present at that site. This process is explained through the following equation:

$$J_{SR}(u^{(1)}, \mathbf{x}, t) = \sum_{\hat{\mathbf{x}} \in \Omega_s} g S_{\hat{\mathbf{x}}}(u^{(1)}, t) \delta(\mathbf{x} - \hat{\mathbf{x}}). \quad (6)$$

The equation models the superposition of calcium injection into the cell at special locations called calcium release units (CRUs), which are modeled as point sources. g controls the amount of calcium injected into the cell and Ω_s represents the set of all CRUs. $S_{\hat{\mathbf{x}}}$ is an indicator function, its value is either 1 or 0 indicating the CRU at $\hat{\mathbf{x}}$ is open or closed. When the CRU is open, it stays open for 5 millisecond, then it remains closed for 100 millisecond. The value of $S_{\hat{\mathbf{x}}}$ is determined by compare the value of a random number and the value of the following probability function

$$J_{\text{prob}}(u^{(1)}) = \frac{P_{\text{max}}(u^{(1)})^{n_{\text{prob}}}}{(K_{\text{prob}})^{n_{\text{prob}}} + (u^{(1)})^{n_{\text{prob}}}}. \quad (7)$$

When the value of the probability function is higher, $S_{\hat{\mathbf{x}}} = 1$, otherwise $S_{\hat{\mathbf{x}}} = 0$. Furthermore, $\delta(\mathbf{x} - \hat{\mathbf{x}})$ denotes a Dirac delta distribution for a CRU located in $\hat{\mathbf{x}}$.

The reaction terms $r^{(i)}$ shown below are nonlinear functions of the different species and couple the three equations.

$$r^{(i)} := \begin{cases} \sum_{j=2}^{n_s} R^{(j)}(u^{(1)}, u^{(j)}), & \text{for } i = 1, \\ R^{(i)}(u^{(1)}, u^{(i)}), & \text{for } i = 2, \dots, n_s, \end{cases} \quad (8)$$

where the reaction rates are given by

$$R^{(i)} = -k_i^+ u^{(1)} u^{(i)} + k_i^- (\bar{u}_i - u^{(i)}) \quad \text{for } i = 2, \dots, n_s. \quad (9)$$

A complete list of the model's parameter values is given in Table 1.

NUMERICAL METHODS

In order to numerically simulate the calcium induced calcium release model, a numerical method must be designed that is very efficient in memory use. The uniform rectangular CRU lattice gives a naturally induced regular numerical mesh, see Figure 1. Additionally, the model uses constant diffusion coefficients. Using a finite element method (FEM) with these properties (constant coefficients, regular mesh) will allow for system matrices whose components can be computed by analytical formulas. Therefore routines, specifically the matrix-vector product, can be designed without an explicitly stored system matrix. A matrix-free method dramatically reduces the memory requirements of the method, thereby making useful computations feasible. This reduced memory requirements also enable us to solve the

Table 1. Table of parameters for the CICR model.

Parameter	Description	Values/Units
t	Time	ms
\mathbf{x}	Position	μm
$u^{(i)}$	Concentration	μM
Ω	Rectangular domain in μm	$(-6.4, 6.4) \times (-6.4, 6.4) \times (-32.0, 32.0)$
$D^{(1)}$	Calcium diffusion coefficient	$\text{diag}(0.15, 0.15, 0.30) \mu\text{m}^2 / \text{ms}$
$D^{(2)}$	Mobile buffer diffusion coefficient	$\text{diag}(0.01, 0.01, 0.02) \mu\text{m}^2 / \text{ms}$
$D^{(3)}$	Stationary buffer diffusion coefficient	$\text{diag}(0.00, 0.00, 0.00) \mu\text{m}^2 / \text{ms}$
$u_{\text{ini}}^{(1)}$	Initial calcium concentration	$0.1 \mu\text{M}$
$u_{\text{ini}}^{(2)}$	Initial mobile buffer concentration	$45.9184 \mu\text{M}$
$u_{\text{ini}}^{(3)}$	Initial stationary buffer concentration	$111.8182 \mu\text{M}$
Δx_s	CRU spacing in x -direction	$0.8 \mu\text{m}$
Δy_s	CRU spacing in y -direction	$0.8 \mu\text{m}$
Δz_s	CRU spacing in z -direction	$0.2 \mu\text{m}$
g	Flux density distribution	$110.0 \mu\text{M} \mu\text{m}^3 / \text{ms}$
P_{max}	Maximum probability rate	$0.3 / \text{ms}$
K_{prob}	Probability sensitivity	$0.2 \mu\text{M}$
n_{prob}	Probability Hill coefficient	4.0
Δt_s	CRU time step	1.0 ms
t_{open}	CRU opening time	5.0 ms
t_{closed}	CRU refractory period	100 ms
k_2^+	Forward reaction rate	$0.08 / (\mu\text{M ms})$
k_2^-	Backward reaction rate	$0.09 / \text{ms}$
\bar{u}_2	Total of bound and unbound indicator	$50.0 \mu\text{M}$
k_3^+	Forward reaction rate	$0.10 / (\mu\text{M ms})$
k_3^-	Backward reaction rate	$0.10 / \text{ms}$
\bar{u}_3	Total bound and unbound buffer	$123.0 \mu\text{M}$
V_{pump}	Maximum pump strength	$4.0 \mu\text{M} / \text{ms}$
K_{pump}	Pump sensitivity	$0.184 \mu\text{M}$
n_{pump}	Pump Hill coefficient	4
J_{leak}	Leak term	$0.320968365152510 \mu\text{M} / \text{ms}$

problem on a fine mesh with relatively small GPU memory. The NVIDIA K20 GPU we use has 5 GB of memory, compared to 64 GB memory connected to CPU. The convergence of the finite element method in presence of measure valued source terms, as they occur in the model (1) and (4), was rigorously shown in [11], and numerical results agree well with the theoretical predictions [2].

The spatial discretization of the three-species application problem with tri-linear nodal finite elements results in a large system of ordinary differential equations (ODEs). This ODE system is solved by the family of numerical differentiation formulas (NDF k) with variable order $1 \leq k \leq 5$ and adaptively chosen time step size [12, 13]. Since these ODE solvers are fully implicit, it is necessary to solve the fully coupled non-linear system of equations at every time step. For its solution a matrix-free newton method is applied, which means that results of the Jacobian-vector products needed in the Krylov subspace method are provided directly without storing the Jacobian. In addition, the usage of the exact Jacobian should lead to quadratic convergence of the Newton method. The linear solver makes use of the iterative BiCGSTAB method with matrix-free matrix-vector multiplies. Table 2 summarizes several key parameters of the numerical method and its implementation. The first three columns show the spatial mesh resolution of $N_x \times N_y \times N_z$, the number of mesh points $N = (N_x + 1)(N_y + 1)(N_z + 1)$, and their associated numbers of unknowns $n_s N$ for the n_s species that need to be computed at every time step, commonly referred to as degrees of freedom (DOF). The following column lists the number of time steps taken by the ODE solver, which are significant and which increase with finer resolutions. The final two columns list the memory usage in GB, both predicted by counting variables in the algorithm and by observation provided in a memory log file produced from the performance run. We notice that even the finest resolution fits comfortably in the memory of one NVIDIA K20 GPU.

CUDA + MPI IMPLEMENTATION

Motivation

The motivations behind General-Purpose Computation on Graphics Processing Unit (GPGPU) are multifold. First of all, the models that we have are becoming more and more complicated. We need to solve the complicated problem with finer meshes, larger (cell) domain, longer simulation time. We also need to prepare for more species which means more PDEs coupled. Furthermore, we might ran into situations that thousands of simulations are required for certain studies [1].

The degree of freedom (DOF) as well as computational burden will increase substantially as the model increases complexity. And the current MPI approach that runs on Multi-core CPU cluster will reach limit. This approach has good scalability on the condition that you have access to large number of compute nodes with cutting edge CPUs.

Hence, offloading to accelerators such as GPUs are considered natural choices. Our problem is suitable for GPU computation because of the following reasons: The program is computationally intensive, heavy computation can be done on the GPU, with few data transfer. The program is also massively parallel, similar tasks are performed repeatedly on different data. Also, from the cost effective aspect, a state-of-the-art GPU card is much cheaper to acquire compare to a up-to-date CPU node, while providing comparable or more throughput.

Hardware Used

The hardware used in the computational studies is part of the UMBC High Performance Computing Facility (HPCF). Each hybrid node contain two eight-core 2.6 GHz Intel E5-2650v2 Ivy Bridge CPUs and 64 GB memory, see Figure 2. Each CPU is connected to one state-of-the-art NVIDIA K20 GPU, see Figure 3. The nodes are connected by a high-speed quad-data rate (QDR) InfiniBand network.

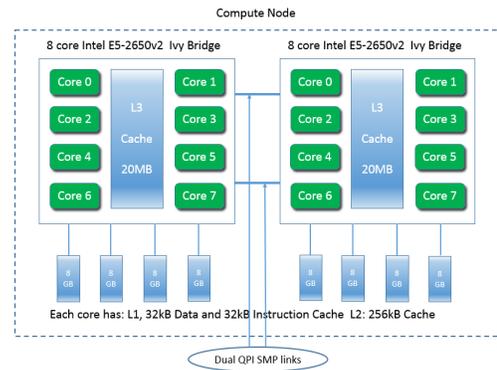


Figure 2. Schematics of CPU: Intel E5-2650v2 Ivy Bridge

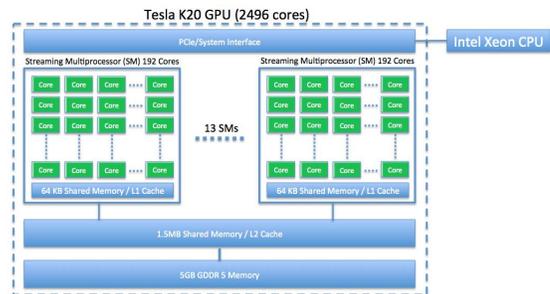


Figure 3. Schematics of NVIDIA Tesla K20 GPU.

CUDA + MPI Workflow

In CUDA programming language, CPU and the system's memory are referred to as host, and the GPU and its memory are referred to as device. Figure 4 explains how threads are grouped into blocks, and blocks grouped into grids. Threads unite into thread blocks – one- two or three-dimensional grids of threads that interact with each other via shared memory and synchpoints. A program (kernel) is executed over a grid of thread blocks. One grid is executed at a time. Each block can also be

Table 2. Sizing study listing the mesh resolution $N_x \times N_y \times N_z$, the number of mesh points $N = (N_x + 1) \times (N_y + 1) \times (N_z + 1)$, the number of degrees of freedom ($\text{DOF} = n_s N$), the number of time steps taken by the ODE solver, and the predicted and observed memory usage in MB for a one-process run.

$N_x \times N_y \times N_z$	N	DOF	number of time steps	memory usage (GB)	
				predicted	observed
$32 \times 32 \times 128$	140,481	421,443	58,416	0.05	0.08
$64 \times 64 \times 256$	1,085,825	3,257,475	73,123	0.41	0.48
$128 \times 128 \times 512$	8,536,833	25,610,499	89,088	3.24	3.68

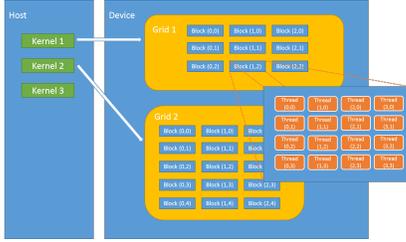


Figure 4. Schematic of Blocks and Threads

one-, two-, or three-dimensional in form. This is due to the fact that GPUs used to work on graphical data, which has 3 dimensions red, green and blue. This now gives much flexibility in launching kernels with different data structure. However, there are still limitations, such as the maximum dimension size of a thread block is (1024, 1024, 64), and the maximum number of threads per block is 1024 for the K20 GPU we have.

The program used to perform the parallel computations presented in this paper is an extension of the one described in [2] and [10], which uses MPI for parallel communications. Therefore, it inherited the main structure of the C program. However, to enable efficient calculations on GPU, almost all calculations have been redesigned to take advantage of GPU parallelism. While inputs are still managed by host, C structs are shared by host and device. Large arrays are allocated directly on device memory before numerical iterations, hence to prevent frequent communications between host and device. Since host handles MPI communication and output to files, data communications between host and device occur before and after these events. The CUDA program has several levels, a detailed discussion is as follows:

The uppermost level is where computational resources are managed. First, MPI processes are setup in `main.cu`. After detect the number of CUDA capable devices (NVIDIA GPUs) on each node, the `main.cu` function then set CUDA device for each MPI process. The idea behind this setup is to allow each MPI process to have access to a unique GPU device. If more than one MPI processes are accessing the same GPU, kernels will queue up and the performance will be degraded. Since in our cluster each GPU enabled node has two GPUs, each connected to a CPU socket via PCI bus, as shown in Figure 3. This means the best approach is to request 2 processes from each node and have each MPI process utilize a unique GPU. We can also run the program in serial,

with a single MPI process and one GPU. The 5 GB GPU memory can hold the memory allocated for all large arrays for our current highest mesh $128 \times 128 \times 512$. After the computational resources are correctly allocated, the `main.cu` program launches the program that does the actual computation. Lastly, the `main.cu` program records the total memory used for each CPU node.

The next level, the program selects the right solver based on a set of parameters. The code is a package that can solve many different problems with one-dimensional, two-dimensional and three-dimensional domains. In the case of the CICR model, `main.cu` call the function `run_parabolic.cu`, Within `problem_par_3d.cu`, we first read parameters from one input file, setup C structs, allow them to be shared by both C and CUDA code, and write arrays associated to the structs. Parameters in Table 1 are read in and setup in these steps. A crucial arrangement in the code is to put C struct type definitions in one h file called `struct_define.h`. After the definition of various struct data types, need to declare each struct with `extern`. Just like we declare all other functions with `extern "C"`, this is allows the mixture of C and CUDA to work properly. This h file is then included by every other `.cu` files. However, to be able to compile correctly, we also need to put regular struct declarations at the beginning of `main.cu`.

At level three, ODE solver is called. The solver is based on numerical differentiation formulas (NDF k) with variable order $1 \leq k \leq 5$ and adaptively chosen time step size, As described in Section , at each time step a nonlinear system is solved via a matrix-free Newton method. The matrix vector multiplication function therein have many GPU kernels that can readily take data already exist on GPU memory, hence to reduce the cost of transferring data between CPU and GPU memories. Cublas is used for dot product with double precision. While running with multiple MPI processes, data transfers between CPU and GPU memories are inevitable. Firstly, data has to be transferred back to CPU memory for output. Secondly, data needs to be transferred to CPU memory before MPI communication, and transferred back to GPU memory for calculation. But, it is expected that the second level of paralism on GPU will outperform CPU. For the matrix-vector multiplication, we split GPU kernels into two parts, one does not require computation on MPI communicated data, the other does. Since non-blocking MPI communication functions such as `MPI_Isend` and `MPI_Irecv` returns immediately, we

can have MPI communication and the execution of first kernel at the same time. We put a `MPI_Waitall` only before the launch of the second kernel, making sure the MPI communication is finished before accessing these data.

Lastly, it is vitally important to design kernels that can run with different mesh size in Table 2, and also have room for higher mesh. The most crucial design in kernels are the choice of block and grid sizes. As mentioned before, each block and grid can have one, two or three-dimensions. This gives much flexibility in launching kernels with different data structure. In the application problem here, the number of threads in one block is determined by the mesh on x-direction N_x , as `dim3 threads(Nx, 1)`. This allows N_x to be as large as the limit of 1024 threads. Moreover the number of blocks in one grid is determined by the mesh on y and z-directions N_y and N_z , as `dim3 blocks(Ny, 1, Nz)`. l means local to the MPI process. `dim3` can be used to define arrays of up to three-dimensions. In this setup, all utility functions and kernels that need to access large arrays on GPU can be designed to use the same block thread counts, making it much easier to program the actual kernels.

RESULTS

Figure 5 shows CRU plots generated from simulations using GPUs, which are similar to those generated by previous C programs run on CPU nodes. The plots in this figure show which CRUs are open at each time step during the simulation. We see that at $t = 100$ a few CRUs are open, the wave mostly spreads along x - and y -dimensions at this point. Later on we see that the CRUs have begun to open on both sides of the cell and spread across it. During our simulation of 1000 ms, several waves have been generated and run across the cell, with similar speed on both ways of the z -direction.

Table 3 summarizes the wall clock times for the CICR problem solved with the finite element method using $p = 1, 2$ and 16 MPI processes on a CPU node with two eight-core CPUs. All runs fit into the memory of the node, but some runs would take longer than the maximum time of 5 days allowed for a job on the system. For the cases, where the run with $p = 1$ process is possible, the parallel scalability is excellent to $p = 2$ processes, and using all 16 cores available on the node is clearly the fastest run in each case.

Table 4 summarizes the wall clock times for the CICR problem solved with the finite element method using a hybrid CPU / GPU node. For mesh resolution $32 \times 32 \times 128$, the wall clock time on one CPU core and one GPU is more than 5 times faster than a serial run on a CPU, but slower than using all 16 cores on a CPU node. For this coarse resolution, the wall clock time on one node with two MPI processes and two GPUs does not improve performance. This is due to time spent on data transfer between CPU and GPU memory dominate over time spent on calculation.

For mesh resolution $64 \times 64 \times 256$, the wall clock time on one CPU core and one GPU is more than 15 times faster than a serial run on a CPU. For this resolution, the wall clock time on one node with two MPI processes and two GPUs is 1.8 times faster than using all 16 cores on a CPU node.

The amount of time needed for calculation increased rapidly due to increased mesh size. For the fine mesh resolution $128 \times 128 \times 512$, the serial run on a CPU is not available due to excessive time requirement. The wall clock time on one node with two MPI processes and two GPUs is around 3 times faster than using all 16 cores on a CPU node.

Table 3. Wall clock time for CICR problem solved with FEM using p MPI processes on a CPU node. E.T. indicates excessive time requirement (more than 5 days).

$N_x \times N_y \times N_z$	$p = 1$	$p = 2$	$p = 16$
$32 \times 32 \times 128$	04:12:42	02:11:30	00:20:28
$64 \times 64 \times 256$	29:39:29	15:33:52	02:36:56
$128 \times 128 \times 512$	E.T.	E.T.	42:07:19

Table 4. CICR problem solved with FEM on a hybrid node using p MPI processes and one GPU per MPI process. Each MPI process launches kernels on a unique GPU. (a) Wall clock time, (b) speedup over $p = 16$ MPI processes on a sixteen-core CPU node.

(a) Wall clock time		
$N_x \times N_y \times N_z$	$p = 1$ 1 GPU	$p = 2$ 2 GPUs
$32 \times 32 \times 128$	00:42:33	00:43:32
$64 \times 64 \times 256$	01:58:19	01:25:32
$128 \times 128 \times 512$	25:09:06	13:46:41
(b) Speedup over $p = 16$ run on CPU Node		
$N_x \times N_y \times N_z$	$p = 1$ 1 GPU	$p = 2$ 2 GPUs
$32 \times 32 \times 128$	0.48	0.47
$64 \times 64 \times 256$	1.33	1.83
$128 \times 128 \times 512$	1.67	3.06

CONCLUSIONS

The results demonstrate that using MPI and CUDA on a hybrid node with two CPUs and two GPUs, the CICR problem can be solved much faster than using all 16 cores of two eight-core GPUs on a CPU node. The data transfer between CPU and GPU memory is inevitable, but can be improved by splitting kernels and use non-block MPI communication. In the future we will investigate possible further improvements like using cuda-aware MPI. Moreover, algorithms that can minimize the data communication without huge sacrifice of accuracy are of interest. Additionally, the performance of several hybrid nodes should be compared to using several CPU nodes.

Acknowledgments

Xuan Huang acknowledges support from the UMBC High Performance Computing Facility (HPCF). The

hardware used in the computational studies is part of HPCF. The facility is supported by the U.S. National Science Foundation through the MRI program (grant nos. CNS-0821258 and CNS-1228778) and the SCREMS program (grant no. DMS-0821311), with additional substantial support from the University of Maryland, Baltimore County (UMBC). See www.umbc.edu/hpcf for more information on HPCF and the projects using its resources.

REFERENCES

1. Brewster, M. W. The Influence of Stochastic Parameters on Calcium Waves in a Heart Cell. Senior thesis, Department of Mathematics and Statistics, University of Maryland, Baltimore County, 2014.
2. Gobbert, M. K. Long-time simulations on high resolution meshes to model calcium waves in a heart cell. *SIAM J. Sci. Comput.* 30, 6 (2008), 2922–2947.
3. Graf, J., and Gobbert, M. K. Parallel performance studies for a parabolic test problem on the cluster maya. Tech. Rep. HPCF-2014-7, UMBC High Performance Computing Facility, University of Maryland, Baltimore County, 2014.
4. Hanhart, A. L., Gobbert, M. K., and Izu, L. T. A memory-efficient finite element method for systems of reaction-diffusion equations with non-smooth forcing. *J. Comput. Appl. Math.* 169, 2 (2004), 431–458.
5. Izu, L. T., Mauban, J. R. H., Balke, C. W., and Wier, W. G. Large currents generate cardiac Ca^{2+} sparks. *Biophys. J.* 80 (2001), 88–102.
6. Izu, L. T., Wier, W. G., and Balke, C. W. Theoretical analysis of the Ca^{2+} spark amplitude distribution. *Biophys. J.* 75 (1998), 1144–1162.
7. Izu, L. T., Wier, W. G., and Balke, C. W. Evolution of cardiac calcium waves from stochastic calcium sparks. *Biophys. J.* 80 (2001), 103–120.
8. Marcotte, C. D., and Grigoriev, R. O. Implementation of pde models of cardiac dynamics on gpus using opencl. *arXiv preprint arXiv:1309.1720* (2013).
9. Pethiyagoda, R., McCue, S. W., Moroney, T. J., and Back, J. M. Jacobian-free newton–krylov methods with gpu acceleration for computing nonlinear ship wave patterns. *Journal of Computational Physics* 269 (2014), 297–313.
10. Schäfer, J., Huang, X., Kopecz, S., Birken, P., Gobbert, M. K., and Meister, A. A memory-efficient finite volume method for advection-diffusion-reaction systems with non-smooth sources. *Numer. Methods Partial Differential Equations* 31, 1 (2015), 143–167.
11. Seidman, T. I., Gobbert, M. K., Trott, D. W., and Kružík, M. Finite element approximation for time-dependent diffusion with measure-valued source. *Numer. Math.* 122, 4 (2012), 709–723.
12. Shampine, L. F. *Numerical Solution of Ordinary Differential Equations*. Chapman & Hall, 1994.
13. Shampine, L. F., and Reichelt, M. W. The MATLAB ODE suite. *SIAM J. Sci. Comput.* 18, 1 (1997), 1–22.

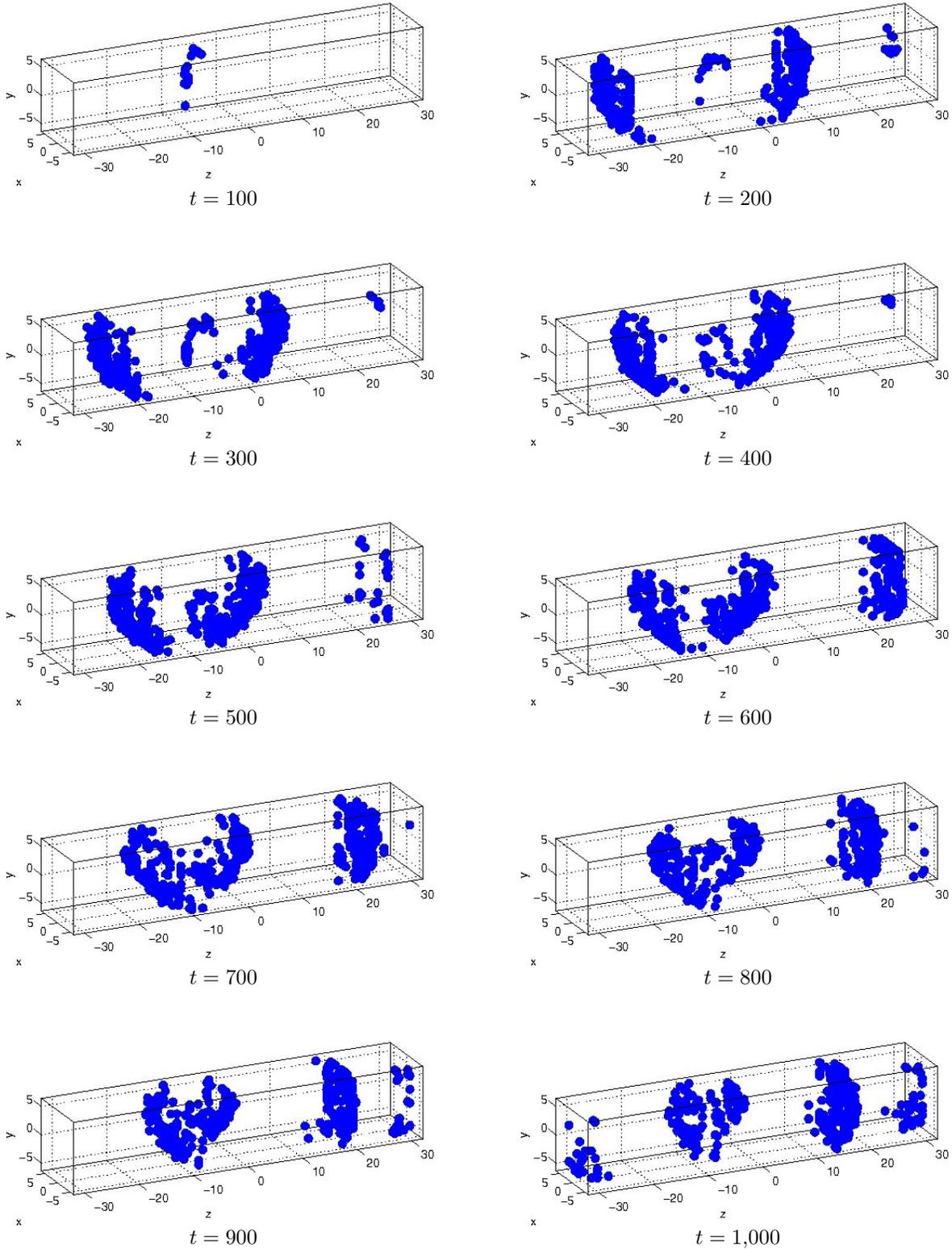


Figure 5. Open calcium release units throughout the cell using finite element method with mesh size $32 \times 32 \times 128$. Based on simulation with CUDA + MPI.

High Performance Kirchhoff Pre-Stack Depth Migration on Hadoop

Chao Li, Yida Wang, Haihua Yan
School of Computer Science and Engineering,
Beihang University
Beijing, China
{hipercomer, wyd, yhh}@cse.buaa.edu.cn

Changhai Zhao, Jianlei Zhang
Research and Development Center, BGP Inc.,
CNPC
Zhuozhou, Hebei, China
{zhaochanghai, zhangjianlei}@cnpc.com.cn

ABSTRACT

Kirchhoff Pre-Stack Depth Migration (KPSDM), a widely used seismic imaging algorithm in petroleum industry, is a typical IO-bound application since a large amount of seismic data and travel timetable data needs to be read from the storage system iteratively during runtime. We present an optimized high performance KPSDM implementation called HKPSDM based on Hadoop where a large I/O aggregated bandwidth is offered cheaply to replace our previous implementation based on Network Attached Storage (NAS) appliances over NFS which have a limited I/O bandwidth when hundreds of processes participate in a migration job. In our implementation, MapReduce facilitates the travel timetable rearrangement and HDFS provides a stable and scalable storage system for seismic data as well as travel timetable data. Various optimizations are applied to HKPSDM to reduce the global bandwidth consumption of HDFS and overhead caused by some disadvantages of HDFS. Experimental results show that HKPSDM can scale better when the number of computing cores ranges from 160 to 800. HKPSDM performs more than 5 times better than NAS-based one when rearranging the travel timetables. And a 37% efficiency improvement is observed when 800 cores are used for migration.

Keywords

Kirchhoff Pre-Stack Depth Migration; Hadoop; MapReduce; HDFS.

INTRODUCTION

Seismic migration aims to obtain the actual image of subsurface with high spatial resolution based on an inversion calculation process in which the energy of seismic waves are rearranged to move dipping reflectors to their true subsurface positions. It is a time-consuming process as the size of input seismic data could achieve dozens or hundreds of terabytes. To shorten the processing time, migration computations are usually carried out on cost-effective clusters with Network Attached Storage (NAS) appliances over NFS.

KPSDM is one of the most important and popular depth-domain imaging algorithms for its ability to process

irregular seismic data and target line computation efficiency [1, 2]. However, KPSDM imposes large I/O workloads [3] on storage system for several reasons listed as follows.

- In travel timetable computation stage, travel timetables of all tracing points need to be rearranged by imaging space, which involves a large amount of I/O operations.
- Besides seismic data, a large amount of travel timetable data needs to be read from the storage system in the migration stage.
- As travel timetables dominate a large part of memory, there is only a small block of imaging space can be migrated for one seismic trace. In addition, integral-based KPSDM involves less computation. More seismic traces need to be fed to computing nodes to balance the computation.

In practice, we find that the huge I/O workloads produced by KPSDM make NAS become the bottleneck for higher scalability. When the number of computing nodes reaches hundreds, performance begins to suffer because of the poor I/O bandwidth of NAS. Besides that, NAS cannot perform well when mapping travel timetables from ray tracing space to imaging space since a large amount of data movement is involved (dozens or hundreds of terabytes worth). Another disadvantage of NAS-based solution is that excessive dominance of the storage system bandwidth by KPSDM will increase risks of impacting end users' experience of interactive jobs and cause throughout degradation when other data-intensive applications are scheduled on the same cluster.

To bypass the bottleneck and reduce side effects on other co-running applications, we implement an optimized KPSDM based on Hadoop [4] called HKPSDM to replace the previous NAS-based one. MapReduce [5] and HDFS [6] are two core components of Hadoop in which MapReduce is a scalable distributed data processing framework with programming simplicity and HDFS is a scalable distributed file system built on the local disks of clusters with high availability. In HKPSDM, MapReduce is utilized to rearrange the travel timetables efficiently according to imaging space. While HDFS is used as a stable and scalable storage system for seismic data and travel timetable data. All processes are divided into groups in which a group

leader is set as a I/O proxy to reduce the bandwidth consumption of HDFS. Another important optimization is caching seismic data acquired from HDFS with local disks to reduce bandwidth consumption further. By pre-fetching data from HDFS, overhead introduced by HDFS's high latency and low bandwidth for single node is covered.

Experimental results of tests running on a production cluster with practical seismic data show that HKPSDM can work stably and efficiently. When applied to rearrange travel timetables, HKPSDM can outperform more than 5 times better than the NAS-based one. Also, HKPSDM scales better when the number of computing nodes ranges from 20 to 100. And a 37% efficiency improvement is observed when 100 computing nodes are used for migration computation.

The rest of the paper is organized as follows. Section 2 introduces the seismic imaging, KPSDM principles, Hadoop framework and related work. Section 3 describes the design and implementation details of HKPSDM. Section 4 demonstrates the experimental results. Section 5 presents the related work and Section 6 concludes the paper.

BACKGROUND AND RELATED WORK

Seismic Imaging

Seismic exploration is widely used to locate the oil and gas reservoir area in petroleum industry by providing the images of earth's interior structures. Seismic imaging can greatly reduce the exploration risks of well drilling which usually costs dozens of million dollars.

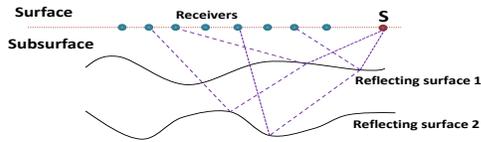


Figure 1: Principle of seismic exploration

Figure 1 describes the principle of seismic exploration. Elastic waves generated at source point S, also called shot point, propagate through the subsurface. With reflection and refraction effects, the waves will travel back to the surface and then be recorded by the geophones. At last, seismic dataset collected by all geophones is processed by a migration algorithm such as KPSDM to obtain the images of earth's subsurface.

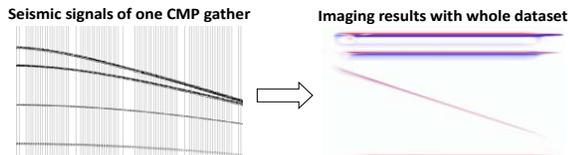


Figure 2: Seismic signals and the imaging results

A trace refers to the recorded seismic waves caused by one shot. It consists of a vector of discrete floating numbers called samples. Each trace contains three important parameters, i.e., the source point, receiver point and the

middle point of source point and receiver point. A gathers is a collection of traces that share common attributes. The notion offset refers to the distance between the source point and the receiver point. A collection of traces that share common middle point, common shot point is called common middle point gather (CMP gather) and common shot gather, respectively.

Figure 2 shows the original seismic signals of one CMP gather and the imaging results resolved by KPSDM algorithm from seismic signals of the whole dataset. The visualization image can clearly illustrate the reflectors of subsurface which provide critical information for geologist to locate potential reservoir areas.

Principle of KPSDM

KPSDM can be expressed by Equation (1), in which ξ denotes the imaging point, $I(\xi)$ denotes the imaging results, $D[t, m, h]$ denotes the effective seismic dataset of ξ , m denotes the common middle point, h denotes the half distance between the source point and the receiver point, also called half offset, and Ω_ξ denotes the migration aperture.

$$I(\xi) = \int_{\Omega_\xi} W(\xi, m, h) D[t = t_D(\xi, m, h), m, h] dm dh \quad (1)$$

$$t_D = t_s + t_r \quad (2)$$

$$\frac{1}{v^2} = \left(\frac{\partial t}{\partial x}\right)^2 + \left(\frac{\partial t}{\partial y}\right)^2 + \left(\frac{\partial t}{\partial z}\right)^2 \quad (3)$$

Figure 3 shows the relationship of ξ and its corresponding dataset D from the view of vertical profile. Migration aperture and the position of ξ delimit valid input range for one imaging point. Any trace whose middle point, source point as well as receiver point all locate within the range contributes to the samples of imaging point ξ . And the function $W(\xi, m, h)$ gives the contribution weighted factor. Thus, for a CMP gather determined by middle point m which is within the valid input range, whether a trace in the gather contributes to image point ξ is determined by its half offset h . This explains the principle behind Equation (1).

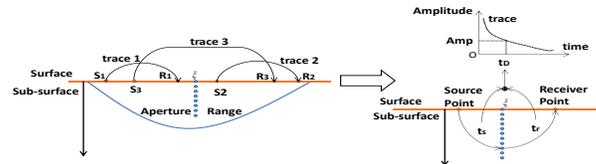


Figure 3: Principle of the KPSDM algorithm

In practice, the process of integral computation is to perform weighted summing calculations on the valid input seismic dataset. Figure 3 illustrates the process how a valid seismic trace contributes to the samples of one imaging point. For one sample of imaging point ξ , the time t_s cost by seismic waves traveling from source point to the sample and t_r cost by seismic waves traveling from the sample to receiver point are need to calculate t_D , the total cost time from source point to receiver point based on Equation (2).

t_D is used to find the corresponding amplitude in the trace which will be added to the current value of the imaging sample as a contribution.

While the travel time t from a source point or receiver point to the imaging sample can be calculated by Formula (3). V denotes the velocity used to characterize the medium through which waves propagate. The value of V is provided by a velocity model. (x, y, z) is the coordination of the imaging sample. Travel time t can be obtained by solving the formula.

Although various parallel implementations of KPSDM exist, generally a migration job can be done in 3 steps [3]:

- Step 1, building travel timetables for imaging space based on ray tracing theory with a velocity model.
- Step 2, preprocessing the input seismic traces.
- Step 3, migrating the seismic traces with travel timetables using KPSDM algorithm and outputting the imaging results to database.

Hadoop Framework

Apache Hadoop is an open sourced software framework. MapReduce and HDFS lie in an essential position of Hadoop. Derived from Google's file system [7], HDFS is built on the local disks attached to computing nodes. Files stored on HDFS are split into 64MB blocks by default. Then these blocks are scattered across the cluster and stored on certain disks. Each block has two extra replications stored on different disks for high availability.

Apache MapReduce, derived from Google's MapReduce framework [5], is a distributed programming and runtime framework built on the top of HDFS. It aims to ease the difficulty of developing efficient distribute programs that can run on large clusters. MapReduce programs can divide into three steps: map, shuffle and reduce. In the map stage, input data is broken into splits logically by a reconfigurable size. These splits are fed into a large pool of mappers in which splits are processed independently and parallel. After enters a mapper, the split is further broken into key-value pairs which will be emitted to the runtime system by map interface. In the shuffle stage, values will be aggregated by keys. And in the reduce stage, values that have a common key will be processed by the reduce interface. MapReduce can take advantage of data locality by moving computation to data to ensure fast I/O speed and high performance. By rewriting the implementation of data splitting, mapping and reducing, programmers can develop their own distributed programs.

DESIGN AND IMPLEMENTATION

Parallelization of KPSDM

In our implementation, the KPSDM algorithm in step 3 mentioned in last Section can be illustrated by Algorithm 1.

In the migration stage, KPSDM needs two types of data, i.e., travel timetable and seismic data. Each data can reach

dozens or hundreds of terabytes. Both of the input data need to be split into small pieces to meet the hardware configuration of each computing node. Based on this parallelization method, KPSDM can be implemented on large high performance clusters with a task-parallel model.

Algorithm 1: KPSDM algorithm in detail

```

foreach offset
  foreach imaging point
    foreach valid input trace
      Get travel timetable vector t1 from source point of
      the trace to the imaging point
      Get travel timetable vector t2 from receiver point
      of the trace to the imaging point
      Accumulate the input sample contribution to the
      samples of the imaging point with t1 and t2
  
```

Thus, parallel tasks of KPSDM are generated by splitting seismic data into segments and imaging space into blocks respectively. As seismic data needs to be cached by local disks, we divide the input seismic data into segments whose size can fit in the available space of local disks. The default threshold is 30GB. The dividing principle of imaging space is that the memory can hold all travel timetable for one imaging block. Note that each segment needs to migrate the whole imaging space. Let M denote the segment number of seismic data, N denote the block number of imaging space. The total number of parallel tasks is $M*N$. The sequential version of KPSDM can be described by Algorithm 2.

Algorithm 2: Sequential KPSDM

```

for  $i = 0$  to  $M$ 
  for  $j = 0$  to  $N$ 
    load travel timetable of  $N_j$  into memory space ttt
    foreach gather in segment  $M_i$ 
      perform migration with gather and ttt
    output image results of  $\langle M_i, N_j \rangle$ 
  
```

For a migration task $\langle M_i, N_j \rangle$, travel timetable of imaging block N_j needs to be loaded into memory first. Then segment M_i is read from storage system as gathers to migrate the imaging block with travel timetable in memory. After all migration computations are done, imaging results for this task will be output into database.

Travel Timetable Rearrangement

Overview

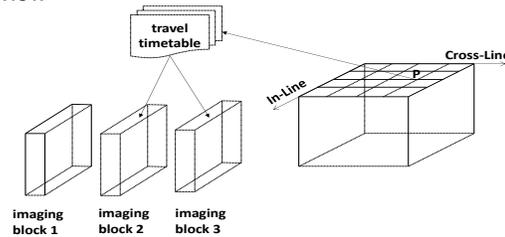


Figure 4: Ray tracing grid and travel timetable of one point

In travel timetable computation stage, the surface of survey area is divided by in-line and cross-line into a two dimensional grid on which ray tracing points are arranged

as shown in Figure 4. Each point of the grid has a travel timetable calculated independently based on ray tracing theory. By treating each point as a task, we employ a master/work architecture to organize all the processes with dynamic task allocation mechanism to calculate all points. Note that travel timetable of one point contains data that belongs to multiple imaging blocks as depicted in Figure 4.

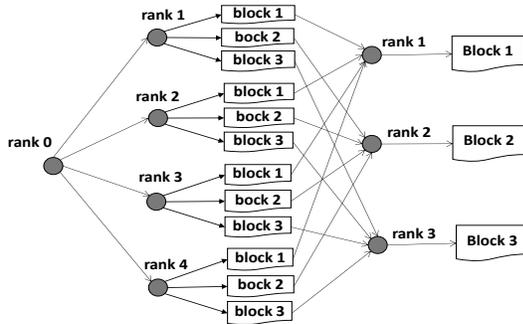


Figure 5: Architecture design of NAS-based solution

A typical design for NAS-based solution is that a process has to maintain a travel timetable file for each imaging block. The travel timetable of one point is split into small parts for different imaging blocks and then each part will be written into the corresponding file. After the travel timetable computation is completed, there is an extra stage where these travel timetable files generated by all worker processes will be merged by imaging block. Figure 5 illustrates the process with 4 worker processes and 3 imaging blocks. A major disadvantage of this scheme is that too many open files may dramatically degrade the performance of computing nodes and NAS. And I/O bandwidth of NAS will be a hinder for merging files efficiently.

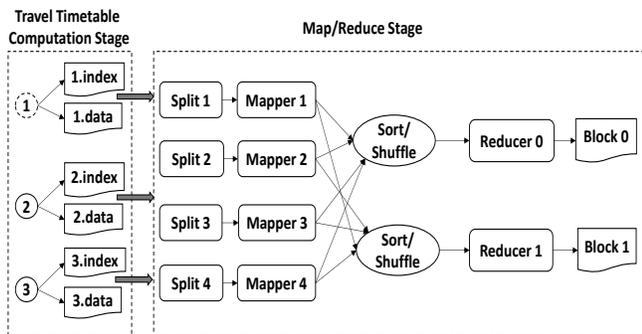


Figure 6: MapReduce based scheme

We utilize MapReduce framework to solve this data-bound problem efficiently. Figure 6 depicts the scheme in which there are 3 workers with 2 imaging blocks. The scheme has two steps: travel timetable computation and map/reduce. In the first step, each process only needs to maintain two files, i.e., a data file and an index file. In the second step, a map/reduce procedure is applied to place the travel timetables that belong to same imaging block together. The rest subsections will provide more information.

Travel Timetable Computation Stage

All travel timetable data of one process is written into the data file stored on HDFS. When a part of travel timetable of one point is written into the data file, an index for that part is necessary to record the relationship between the part and its responding imaging block. The index at least should contain information as follows.

- Imaging block ID, which will be used as a key for Map/Reduce.
- Length of the part data, as each part is compressed before writing out, the length of the part is variable. This is important for marking the boundary between current part and next part so that a unit that contains the index and data can be identified by Map/Reduce.

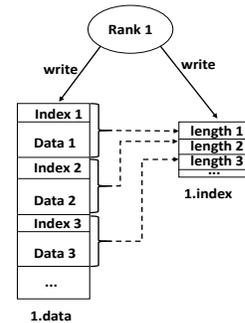


Figure 7: Layout of the data file and index file

A critical optimization is an extra independent index file that records the length of each unit. Figure 7 illustrates the layout of the data file and index file. Before Map/Reduce starts, input data needs to be divided into splits. Although it is feasible to construct all splits by reading all data files on HDFS, doing so is very time-consuming since the size of all data can achieve dozens of terabytes. With the index file, the splitting process can be much more efficient since the data that needs to be read from HDFS is very small. When a unit is written to the data file, the length of the unit is written to the index file at the same time as shown in Figure 7.

Map/Reduce Stage

By rewriting getSplits interface of class FileInputFormat provided by Hadoop, we can implement data splitting for travel timetable data. Algorithm 3 illustrates the splitting process. The element of array *files* contains the information about the data file and index file. Array *splits* maintains the splits divided from all data files. *splitSize* is 64MB by default. *startPos* records the start offset of the split in the data file and *curLen* is the size of the split.

Algorithm 3: Splitting Travel Timetable Data

```

for fileId = 0 to files.length:
    dataFile = files[fileId].dataFile
    indexFile = files[fileId].indexFile
    startPos = 0, unitSize = 0, curLen = 0
    inputStream = open(indexFile)

```

```

while ( ! inputStream.end() ) :
    unitSize = inputStream.readInt()
    if(curLen + unitSize > splitSize) {
        splits.add ( new Split ( dataFile.getPath() ,
            startPos, curLen))
        startPos += curLen; curLen = unitSize
    } else {
        curLen += unitSize
    }
}
if(curLen > 0)
    splits.add ( new Split ( dataFile.getPath() , startPos ,
        curLen ))
inputStream.close()
}

```

Another advantage of Algorithm 3 is that any unit must be whole in one split, which will greatly simplify the generation of key-value pairs after the splits is fed to a mapper. Algorithm 4 presents how to parse key-value pairs from a split. *pos* and *end* is the start and end offset of the split in the data file respectively. *indexLen* is the size of the index. Imaging block ID is set as the key and value is the whole unit. After the key-value pair enters the map interface, it will be emitted out to the runtime system. We rewrite the partition interface to make *k*-th reducer receive all the values associated with key *k*. And the reducer number is set to be the number of imaging blocks. When reduce interface receives the key-value pair, only the value will be written to the file which is corresponding to the imaging block.

Algorithm 4: Parse Next Key-Value Pair

```

if ( pos < end ) {
    blkId = inputStream.readInt();
    len = inputStream.readInt();
    inputStream.seek(-8, SEEK_CUR);
    inputStream.read(buf, indexLen + len);
    key.set(blkId);
    value.set(buf, 0, indexLen + len);
    pos = pos + indexLen + len;
} else {
    processed = true;
}

```

Migration Framework

A two-layer master/worker architecture is employed to organize all processes as shown in Figure 8. Process 0 is set as root whose responsibility includes distributing migration tasks and outputting imaging results into database. The rest processes are divided into groups in which the one with least rank is set to be leader and all members in the group are workers. By default, there are four members in one group.

Before migration starts, root will build a task pool containing N migration tasks for each segment. Once migration starts, group leader will ask root for a segment ID. Then the group is responsible for migrating the segment to the whole imaging space. To reduce the I/O pressure on the HDFS, migration for one segment in one group has two steps as follows.

- In the first step, after all members have finished loading the travel timetable into memory, group leader begins to read gathers from HDFS and broadcasts them to members. Worker who receives the gather will migrate it to the imaging space first and then write it into local disk. After first step finishes, the whole segment will be buffered on the local disk.
- In the second step, the next N-4 migration task will be done by members in the group. The key point is that each member can get gathers of the segment from its own local disk.

For one group, the seismic segment only needs to be read once from HDFS and all N tasks can be migrated. That greatly reduces the bandwidth consumption of HDFS on acquiring seismic data. The bandwidth of HDFS is mainly used to provide travel timetable data.

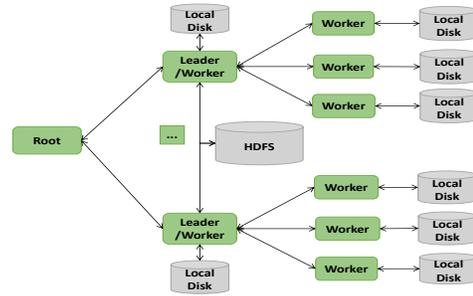


Figure 8: Architecture of all processes

Pre-fetching Mechanism

As Hadoop is optimized for high throughput at the expense of latency [4], comparing with NAS, there is a disadvantage of low bandwidth and high latency for each computing node. We adopt a pre-fetching mechanism to reduce the overhead introduced by this disadvantage.

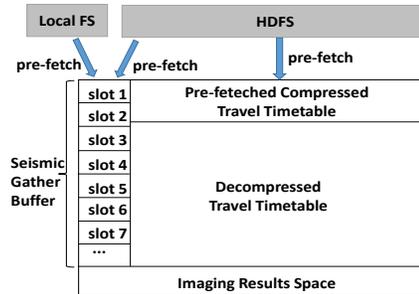


Figure 9: Pre-fetching mechanism

Figure 9 illustrates the memory layout for KPSDM. There is memory space for current decompressed travel timetable and pre-fetched next compressed travel timetable. As long as the time cost on migrating current imaging block can cover time cost on reading travel timetable for next imaging block, the overhead can be eliminated. A seismic buffer is allocated for buffering the acquired gathers from storage system. Overhead can also be reduced by pre-fetching seismic gathers into seismic buffer.

EXPERIMENTAL EVALUATION

HKPSDM is evaluated on hp1, a gigabit Ethernet cluster containing 110 computing nodes. Each node is equipped with 8GB memory, 2 Intel(R) Xeon(R) E5430 2.66GHz CPU with 8 cores in total. One global 15TB NFS file system is available for all nodes. Each node has a local disk in size of 276GB and runs Red Hat Enterprise Linux Server release 6.2 operating system. We use practical seismic data collected from a field located in western China for all tests. The ray tracing grid of the migration job is 4x4 with 75680 points in total and the output imaging grid is 40x1. The seismic data size is 125GB and is split into 5 segments. The travel timetable data size is 79GB and is split into 139 blocks. All optimizations in migration stage of HKPSDM are also applied to the NAS-based scheme for contrast fairness.

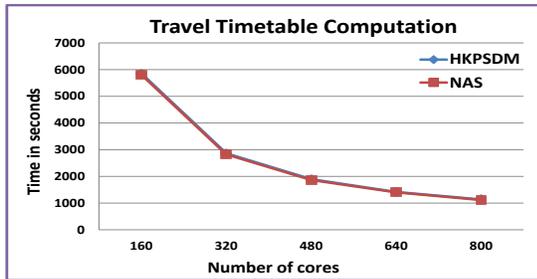


Figure 10: Travel timetable computation

First, we contrast the performance between HKPSDM and NAS-based KPSDM. In each test, we range the cores number from 160 to 800, i.e., nodes number ranges from 20 to 100. When nodes number changes, we reconfigure the Hadoop. Figure 10, Figure 11 and Figure 12 illustrate the time cost on travel timetable computation, travel timetable rearrangement and migration, respectively.

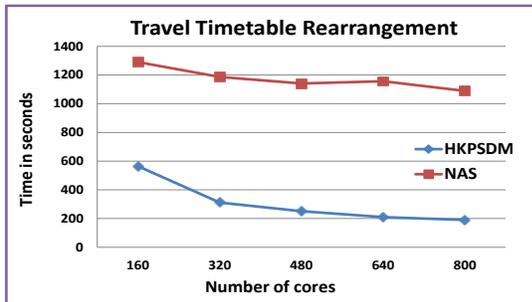


Figure 11: Travel timetable rearrangement

Figure 10 shows that for compute-intensive travel timetable computation stage, HKPSDM performs equally well with NAS-based KPSDM since the time cost by two solutions are very close.

For data-intensive travel timetable rearrangement, HKPSDM performs significantly better as shown in Figure 11. Since the total data throughput of travel timetable rearrangement for NAS is certain, the time cost on this stage is around 1200s for different number of nodes. While

Hadoop provides an increasing I/O bandwidth and computing power with the increasing number of nodes, the performance can scale far better than NFS. For example, HKPSDM performs more than 5 times better than NFS-based solution when using 100 nodes.

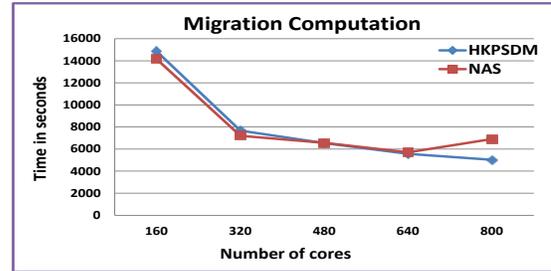


Figure 12: Migration computation

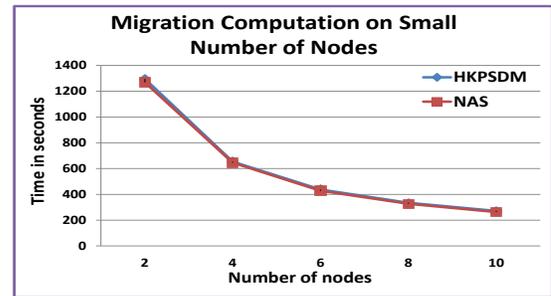


Figure 13: Migration with small number of nodes

The migration stage of KPSDM is memory-intensive. Figure 12 reports that when nodes number ranges from 20 to 60, HKPSDM performs slightly worse than NAS-based version since Hadoop needs to occupy a certain amount of system resources such as CPU, memory and even cache to buffer data, send data to other nodes and receive data from other nodes, etc.

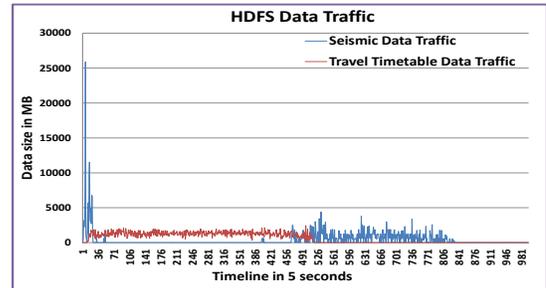


Figure 14: HDFS data traffic monitor for HKPSDM

We also conduct a migration performance contrast experiment on smaller amount of nodes whose number ranges from 2 to 10 as shown in Figure 13. The input data is reduced to 1.5GB and only one in-line is produced. Basically, the results displayed in Figure 13 is similar with Figure 12 when nodes number ranges from 20 to 60. A disadvantage of NAS is that when too many I/O requests need to be processed, total throughput suffers. That is the reason why time cost on migration with 100 nodes is even longer than 80 nodes as shown in Figure 12.

As group leaders need to read seismic data from storage system at start of migration stage, with an increasing number of leaders, I/O bandwidth of NAS becomes the bottleneck. However, with bandwidth scalability, HKPSDM outperforms NFS-based KPSDM when more nodes adopted. And there is a 37% efficiency improvement comparing with NAS-based KPSDM when 100 computing nodes are used for migration computation.

Figure 14 shows the seismic data traffic and travel timetable data traffic of HKPSDM when using 100 nodes. Once migration starts, 99 nodes begin to load travel timetable from HDFS, the average I/O bandwidth of HDFS for reading travel timetable at this stage can achieve about 1 GB/s. When first round of migration starts, HDFS can offer seismic data traffic stably at the speed of 250 MB/s approximately. After first round finishes, benefiting from local disk caching mechanism, all seismic data will be read from local disks. Only travel timetable data needs to be read from HDFS and a smooth travel timetable data traffic can be offered at the speed of 200 MB/s approximately by HDFS.

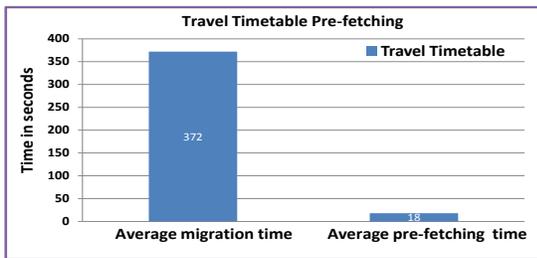


Figure 15: Travel timetable pre-fetching

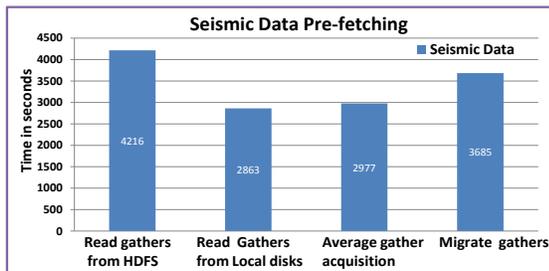


Figure 16: Seismic data pre-fetching

Figure 15 and Figure 16 show results of data pre-fetching optimization. For travel timetable, as migration time for one imaging block is far longer than pre-fetching time, so overhead introduced by acquiring travel timetable data from HDFS can be ignored. Nevertheless, for seismic data, time cost on reading a gather from HDFS is longer than time cost on migrating it. However, the overhead can still be reduced by seismic gather pre-fetching mechanism. Figure 16 also reports that reading a gather from local disk costs less time averagely than migrating it, again demonstrating the effects of local disk caching mechanism.

To assess the accuracy of the seismic migration results produced by HKPSDM, we compute error ratio under

different relative error thresholds as shown in Table 1. All relative errors are less than 10^{-3} and about 0.03% imaging results have larger relative errors than 10^{-4} . In practice, these errors are acceptable as no difference can be perceived between the images produced by two different implementations.

Relative errors	$\leq 1e-7$	$\leq 1e-6$	$\leq 1e-5$	$\leq 1e-4$	$\leq 1e-3$
Error ratio %	62.60	78.16	96.24	99.97	100

Table 1: Assessment of the seismic migration accuracy

Table 2 shows the results of the optimization on speeding up the travel timetable splitting. With an index file, the size of data needs to be read from HDFS can be reduced to 1/2300~1/12000. Comparing with the time cost on the migration and travel timetable computation, time cost on splitting is negligible.

Ray tracing grid	Data size (GB)	Index size (MB)	Ratio	Time (s)
3x3	156	68	2349	11
4x4	79	22	3677	9
10x10	11	0.87	12947	2

Table 2: Time cost on splitting travel timetable

RELATED WORK

Several other seismic migration algorithms such as Reverse Time Migration(RTM) [8], Kirchhoff Prestack Time Migration (KPSTM) [1] and Gaussian Beam Migration (GBM) [9] may be used for seismic migration. Based on our experience obtained from completed migration projects, KPSTM, KPSDM and RTM are three most popular migration algorithms currently. Generally, RTM is more time-consuming than KPSDM and KPSTM for its high computation intensity. And KPSDM can produce imaging results with higher accuracy than KPSTM.

Through each migration algorithm mentioned above has several parallelization implementations in the state of art. The common characteristic of this problem class is that each algorithm needs to acquire large scale of data such as seismic traces from the storage system constantly. As the modern high performance clusters evolve quickly with more computing nodes, more cores, and more powerful hardware accelerators including GPU, MIC, FPGA, etc., the I/O/flops of the clusters are decreasing. The migration algorithms deployed on these clusters may have to face the I/O bottleneck problems. Since KPSDM has larger I/O workloads comparing with other migration algorithms caused by less computations and extra input data, i.e., the travel timetable, the scalability of KPSDM has been affected by the I/O bottleneck in practice. We propose a

MapReduce-based scheme to solve the travel timetable rearrangement efficiently and a HDFS-based KPSDM implementation with several optimizations to bypass the I/O bottleneck. We believe that the Hadoop-based scheme can also help to solve the potential I/O bottleneck problems for other migration algorithms.

An I/O bottleneck problem that affects the scalability of RTM is reported in [10]. And Kirchhoff Pre-Stack Time Migration (KPSTM) is implemented based on MapReduce in [11]. A MapReduce implementation of KPSDM on GPU grid [12] is probably closest to our study. However, we focus on the performance of KPSDM. Since Hadoop cannot support iterative computation well [13] and the migration stage of KPSDM involves a large amount of iterations, we choose to use MapReduce to rearrange travel timetables and HDFS as a storage system.

Several commercial parallelization implementations of KPSDM algorithm use the similar method as proposed in [14]. The main defect is that the method cannot scale on the problem domain when the memory of all computing nodes cannot hold the travel timetable data that is corresponding to one in-line. In [15], all computing nodes acquire seismic data by reading the shared storage system directly. The main problem of this scheme is that huge I/O workloads are imposed on the storage system which will affect the scalability seriously. Process groups migration are proposed in [16]. Nevertheless, the side effect of synchronization caused by seismic traces broadcast within a process group leading to a large amount of CPU cycles being wasted. The parallelization algorithm we used combined with several optimizations can avoid these problems.

CONCLUSION

In this paper, we present a Hadoop-based high performance implementation of KPSDM which is usually deployed on clusters with NAS over NFS. The components of Hadoop we used in our scheme are HDFS and MapReduce. MapReduce allows travel timetable rearrangement to be done efficiently. HDFS provides a stable storage system for seismic data and travel timetable data. With a series of optimization techniques, experimental results show that HKPSDM can run stably and efficiently. The results is especially encouraging when considering that HDFS can be used as a very cost-effective storage system to handle the increasingly disparity of the I/O capacity and computing performance of modern high performance clusters.

REFERENCES

1. Yilmaz, Ö., & Doherty, S. M. (1987). Seismic data processing (Vol. 2). Tulsa: Society of Exploration Geophysicists
2. Bevc D. Imaging complex structures with semirecursive Kirchhoff migration[J]. Geophysics. 1997, 62(2): 577-588.
3. Teixeira, D., Yeh, A., & Sampath Gajawada, T. G. S. (2013). Implementation of Kirchhoff prestack depth migration on GPU. Expanded Abstracts of 83rd Annual Internat SEG Mtg, 2013:3683-3686.
4. White, T. (2009). Hadoop: the definitive guide: the definitive guide. " O'Reilly Media, Inc."
5. Dean, J., & Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. Communications of the ACM, 51(1), 107-113.
6. Shvachko, K., Kuang, H., Radia, S., & Chansler, R. (2010, May). The hadoop distributed file system. In Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on (pp. 1-10). IEEE.
7. Ghemawat, S., Gobioff, H., & Leung, S. T. (2003, October). The Google file system. In ACM SIGOPS Operating Systems Review (Vol. 37, No. 5, pp. 29-43). ACM.
8. Baysal, E., Kosloff, D., and Sherwood, J.W.C., 1983, Reverse time migration: Geophysics, 48, 1514-1524
9. Hill, N. R. (1990). Gaussian beam migration. Geophysics, 55(11), 1416-1428.
10. Perrone, M., Zhou, H., Fossum, G., & Todd, R. (2010, June). Practical VTI RTM. In 72nd EAGE Conference & Exhibition.
11. Rizvandi, N. B., Bolori, A. J., Kamyabpour, N., & Zomaya, A. Y. (2011, October). MapReduce implementation of prestack Kirchhoff time migration (PKTM) on seismic data. In Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2011 12th International Conference on (pp. 86-91). IEEE.
12. Zhang, J., Xu, W., Meng, X., Li, J., & Yang, Q. (2013, June). A Data-ware Scheduling Framework on GPU Grid for Kirchhoff Prestack Depth Migration. In 75th EAGE Conference & Exhibition incorporating SPE EUROPEC 2013.
13. Bu, Y., Howe, B., Balazinska, M., & Ernst, M. D. (2010). HaLoop: Efficient iterative data processing on large clusters. Proceedings of the VLDB Endowment, 3(1-2), 285-296.
14. Dai, H. (2005). Parallel processing of prestack Kirchhoff time migration on a PC cluster. Computers & geosciences, 31(7), 891-899.
15. Chang, H., VanDyke, J. P., Solano, M., McMechan, G. A., & Epili, D. (1998). 3-D prestack Kirchhoff depth migration: From prototype to production in a massively parallel processor environment. Geophysics, 63(2), 546-556.
16. Li, J., Hei, D., & Yan, L. (2009, August). Partitioning Algorithm of 3-D Prestack Parallel Kirchhoff Depth Migration for Imaging Spaces. In Grid and Cooperative Computing, 2009. GCC'09. Eighth International Conference on (pp. 276-280). IEEE.

Parallel QR algorithm for the C-method: application to the diffraction by gratings and rough surfaces

Cihui Pan

Laboratoire PRiSM,
45 avenue des Etats-Unis,
78035 Versailles, France.
Laboratoire LATMOS,
11 Boulevard d'Alembert,
78280 Guyancourt, France.
Laboratoire Maison de la
Simulation,
Bâtiment 565, 91191
Gif-sur-Yvette cedex, France.
cihui.pan@prism.uvsq.fr

Nahid Emad

Université de Versailles
Saint-Quentin en Yvelines,
laboratoire PRiSM,
45 avenue des Etats-Unis,
78035 Versailles, France.
laboratoire Maison de la
Simulation,
Bâtiment 565, 91191
Gif-sur-Yvette cedex, France.
Nahid.Emad@prism.uvsq.fr

Richard Dusséaux

Université de Versailles
Saint-Quentin en Yvelines,
laboratoire LATMOS,
11 Boulevard d'Alembert,
78280 Guyancourt, France.
richard.dusseaux@latmos.ipsl.fr

ABSTRACT

The curvilinear coordinate method (C-method) is an exact method for analysing of electromagnetic waves scattering from rough surfaces. It is based on Maxwell's equations under covariant form written in a non-orthogonal coordinate system. This method leads to an eigenvalue system. All the eigenvalues and eigenvectors of the scattering matrix are required. The QR algorithm seems to be a suitable solution for this high dimension, dense, non-symmetric and complex scattering matrix. In this paper, we present the parallel QR algorithm that is specifically designed for the C-method. We define the "early shift" for the scattering matrix according to the property that we have observed. We mixed the "early shift", Wilkinson's shift and exceptional shift together to accelerate the convergence. Especially, we use the "early shift" first in order to have quick deflation of the real eigenvalues of the scattering matrix. The multi-window bulge chain chasing and parallel aggressive early deflation are used. The multi-window bulge chain chasing approach ensures that most computations are performed in 3 BLAS operations. The aggressive early deflation approach can detect deflation much quicker and thus accelerate convergence. Mixed MPI-OpenMP techniques are utilized for performing the codes to distributed memory platforms. Numerical experiments are performed and applications of this parallel QR algorithm are applied to a real physical problem of diffraction.

Author Keywords

Electromagnetism; parallel QR; shift algorithm; multishift; aggressive early deflation.

ACM Classification Keywords

D.1.3 Programming techniques : Parallel programming; I.6 Simulation and modeling: Miscellaneous

INTRODUCTION

Scattering of electromagnetic waves by rough surfaces has aroused the interest of physicists and engineers for many years because of its wide range of applications in optics, material sciences, communications, oceanography and remote sensing. The analysis of rough surfaces with parameters close to the incident light wavelength requires a rigorous vectorial formalism. The C-method is one of the most efficient and versatile theoretical tools for analysing periodic surfaces, i.e. gratings and rough surfaces [1, 2, 3]. It is based on Maxwell's equations solved in a non-orthogonal coordinate system fitted to the grating geometry. Discretizing the Maxwell's equations under the non-orthogonal coordinate system and separating variables lead to solving the eigenvalue problem of the high dimension, dense and non-symmetric scattering matrix. The scattered field is expanded as a linear combination of eigen-solutions satisfying the outgoing wave condition. The boundary conditions allow the diffraction amplitudes to be determined.

On the one hand, the C-method has a great demand for solvers of large-scale eigenvalue problems. Although iterative eigen-solvers, such as Conjugate Gradient methods, Krylov subspace methods or Jacobi-Davidson methods have been developed to deal with such large-scale problem, they have the possibility of missing some eigenvalues, which are very dangerous in application and will cause the whole C-method fail. So standard iterative methods are ineffective for the C-method because all the eigenvalues and eigenvectors are needed. In contrast, the QR algorithm, which is based on similarity transformations, calculates all the eigenvalues and eigenvectors without any danger of missing some eigen-solutions.

On the other hand, the C-method and the physical interpretation behind the C-method provides very good approximation to all the real eigenvalues before any calculations. This provides the possibility of quick deflation. A new shift strategy is needed. So the C-method requires a specifically designed parallel QR algorithm.

In this article, we present the parallel QR algorithm that is specifically designed for the C-method. We defined the “early shift” for the scattering matrix according to the property that we have observed. We mixed the “early shift”, Wilkinson’s shift and exceptional shift together to accelerate the convergence. Especially, we use the “early shift” first to have quick deflation of the real eigenvalues of the scattering matrix. The multi-window bulge chain chasing and parallel aggressive early deflation are used. Mixed MPI-OpenMP techniques are utilized for performing the codes to distributed memory platforms. Numerical experiments are performed and applications of this parallel QR algorithm are applied to a real physical problem of diffraction .

The paper is structured as follows. In section 2, we present the C-method. In section 3, we introduce the specifically designed parallel QR algorithm for the C-method. In section 4, we present the implementation details, architectures and platforms. In section 5, numerical experiments are performed and results are discussed.

PRESENTATION OF THE PROBLEM

As shown in Figure 1, a surface by equation $y = a(x)$ separates two different media. It is illuminated by a monochromatic plane wave with wavelength λ . The incident wave vector \vec{k}_i is defined by the incident angle θ .

$$\vec{k}_i = \alpha_0 \vec{u}_x + \beta_0 \vec{u}_y \quad (1)$$

with $k = \frac{2\pi}{\lambda}$, $\alpha_0 = k \sin \theta$, $\beta_0 = k \cos \theta$. The surface could be periodic or non periodic. Here we consider periodic surfaces or periodic random surfaces. We represent the vector function

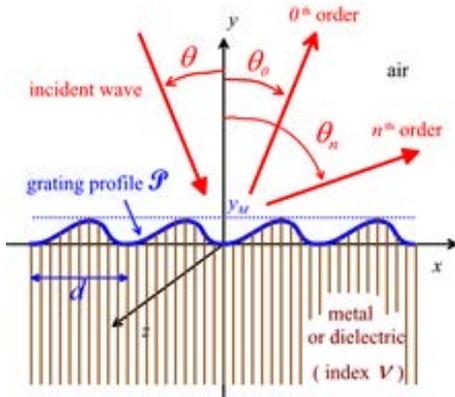


Figure 1: Diffraction of electromagnetic plane wave from rough surface

by its complex vector function and omit its time-dependence factor $\exp(j\omega t)$. So for the horizontal ($E_{//}$) polarization and

vertical ($H_{//}$) polarization,

$$F_0(x, y) = \exp(-j\alpha_0 x + j\beta_0 y) = \begin{cases} E_{0,z}(x, y) & E_{//} \\ Z_1 H_{0,z}(x, y) & H_{//} \end{cases} \quad (2)$$

and

$$Z_1 \vec{H} = \frac{\vec{k}_i}{k} \wedge \vec{E} \quad (3)$$

where $Z_1 = 120\pi$.

The reflected and transmitted plane waves can be written in a similar form. But, for rough surface, we have, in addition to the incident, reflected and transmitted plane waves, a scattered field $F(x, y)$ because of the deformation. The problem consists in working out the scattered field within the two media. The rough surface here is generated by simulation.

We consider a new coordinate system:

$$(x', y' = u, z') = (x, y - a(x), z) \quad (4)$$

The Maxwell’s equations associated with the constitutive relations are then written as

$$\begin{cases} \frac{j}{k_1} \frac{\partial F(x, u)}{\partial u} = \frac{j}{k_1} b(x) \frac{\partial F(x, u)}{\partial x} + c(x) G(x, u) \\ \frac{j}{k_1} \frac{\partial G(x, u)}{\partial u} = \frac{1}{k_1^2} \frac{\partial}{\partial x} (c(x) \frac{\partial F(x, u)}{\partial x}) + \nu^2 F(x, u) \\ + \frac{j}{k_1} \frac{\partial}{\partial x} (b(x) G(x, u)) \end{cases} \quad (5)$$

with

$$b(x) = \frac{\frac{da}{dx}}{1 + (\frac{da}{dx})^2}, c(x) = \frac{1}{1 + (\frac{da}{dx})^2}$$

and in medium i , $F(x, u) = F_i(x, u)$, $G(x, u) = G_i(x, u)$, $\nu = \nu_i$, $i = 1, 2$. In $E_{//}$ polarization, $F(x, u) = E_{z'}(x, u)$, $G(x, u) = Z_1 H_{x'}(x, u)$. In $H_{//}$ polarization, $F(x, u) = \frac{Z_1}{\nu} H_{z'}(x, u)$, $G(x, u) = -\nu E_{x'}(x, u)$.

System (5) can be written in the form as follows:

$$\frac{j}{k_1} \frac{\partial \psi(x, u)}{\partial u} = \mathcal{L} \psi(x, u) \quad (6)$$

with

$$\mathcal{L} = \begin{pmatrix} \frac{j}{k_1} b(x) \frac{\partial}{\partial x} & c(x) \\ \frac{1}{k_1^2} \frac{\partial}{\partial x} (c(x) \frac{\partial}{\partial x}) + \nu^2 & \frac{j}{k_1} \frac{\partial b(x)}{\partial x} \end{pmatrix} \quad (7)$$

and

$$\psi(x, u) = \begin{pmatrix} F(x, u) \\ G(x, u) \end{pmatrix} \quad (8)$$

We separate the variables by writing $\psi(x, u) = \varphi(x) \kappa(u)$, then we get

$$\frac{j}{k_1 \kappa(u)} \frac{d\kappa(u)}{du} = \frac{\mathcal{L} \varphi(x)}{\varphi(x)} = r = \text{constant} \quad (9)$$

So we conclude that $\kappa(u) = C \exp(-jk_1 r u)$, $\mathcal{L} \varphi(x) = r \varphi(x)$ and $\psi(x, u) = A \exp(-jk_1 r u) \varphi(x)$. If the function

$a(x)$ is a period function with d its period, then one has,

$$\begin{cases} a(x) = \sum_m a_m \exp(-j2\pi m x/d) \\ f(x) = \sum_m f_m \exp(-j\alpha_m x) \\ g(x) = \sum_m g_m \exp(-j\alpha_m x) \end{cases} \quad (10)$$

with $\varphi(x) = \begin{pmatrix} f(x) \\ g(x) \end{pmatrix}$ and $\alpha_m = k_1 \sin\theta + m \frac{2\pi}{d}$. Under this function decomposition, the eigenproblem $\mathcal{L}\varphi(x) = r\varphi(x)$ has a matrix form as follows:

$$[\mathcal{L}]\vec{\varphi} = r\vec{\varphi} \quad (11)$$

with

$$[\mathcal{L}] = \begin{pmatrix} [\mathcal{L}_{ff}] & [\mathcal{L}_{fg}] \\ [\mathcal{L}_{gf}] & [\mathcal{L}_{gg}] \end{pmatrix} \text{ and } \vec{\varphi} = \begin{pmatrix} \vec{f} \\ \vec{g} \end{pmatrix} \quad (12)$$

where $[\mathcal{L}_{ff}] = [C][\dot{A}][\tilde{\alpha}]$, $[\mathcal{L}_{fg}] = [C]$, $[\mathcal{L}_{gf}] = \nu^2[I] - [\tilde{\alpha}][C][\tilde{\alpha}]$, $[\mathcal{L}_{gg}] = [\tilde{\alpha}][C][\dot{A}]$, $[\dot{A}]_{pq} = \dot{a}_{p-q} = (p - q) \frac{2\pi}{d} a_{p-q}$, $[I]_{pq} = \delta_{p-q}$, $[C] = ([I] + [\dot{A}][\dot{A}])^{-1}$, $\tilde{\alpha}_p = \frac{\alpha_p}{k_1}$, $[\tilde{\alpha}]_{pq} = \delta_{p-q} \tilde{\alpha}_p$, $(\vec{f})_p = f_p$, $(\vec{g})_p = g_p$, $\forall (p, q) \in \mathbb{Z}^2$. Equations (11) and (12) give an eigenvalue system of infinite dimension. In a numerical computation, one can truncate it to a finite order problem with a truncation order M . The new system is similar to the original one except that now we have $-M \leq p, q \leq M$.

By solving the truncated eigenvalue problem, one gets:

$$\begin{cases} F_n(x, u) = f_n(x) \exp(-jk_1 r_n u) \\ = \sum_{-M \leq m \leq M} f_{mn} \exp(-j\alpha_m x) \exp(-jk_1 r_n u) \\ G_n(x, u) = g_n(x) \exp(-jk_1 r_n u) \\ = \sum_{-M \leq m \leq M} g_{mn} \exp(-j\alpha_m x) \exp(-jk_1 r_n u) \end{cases} \quad (13)$$

So we are left with the eigenproblem of order $4M+2$. Finally, the field can be represented as a linear combination of all the solutions that verifies the outgoing conditions.

$$\psi^{(i)}(x, u) = \sum_{n=1}^{2M+1} C_n^{(i)} \psi_n^{(i)}(x, u), i = 1, 2 \quad (14)$$

and the amplitudes C_n are determined by solving the boundary conditions at $u = 0$ (i.e., at $y = a(x)$). The boundary conditions stipulate the continuity of the electric and magnetic components parallel to the surface. We consider the lossless medium, i.e. the medium with optical index ν real. The power balance criterion[4]

$$\sum_n \epsilon_n^{(1)} + \sum_n \epsilon_n^{(2)} = 1 \quad (15)$$

where $\epsilon_n^{(i)} = |C_n^{(i)}|^2 \frac{\cos\theta_n^{(i)}}{\cos\theta}$, $\theta_n^{(i)} = a \sin(\tilde{\alpha}_n^{(i)})$, $i = 1, 2$ is checked to see if the truncation order M is large enough.

ALGORITHMS

The idea of the QR algorithm is to compute the Schur decomposition of a matrix $A \in \mathbb{C}^{n \times n}$:

$$T = Z^H A Z \quad (16)$$

by iterating the QR decomposition [5]. Here $T \in \mathbb{C}^{n \times n}$ is a upper triangular matrix. Formally, let $A_0 := A$, at the k -th step (starting with $k = 0$), we compute the QR decomposition $A_k = Q_k R_k$ where Q_k is an unitary matrix and R_k is an upper triangular matrix. We then form $A_{k+1} = R_k Q_k$. Note that

$$A_{k+1} = R_k Q_k = Q_k^H Q_k R_k Q_k = Q_k^H A_k Q_k = Q_k^{-1} A_k Q_k, \quad (17)$$

so all the matrices A_k are similar to each other and hence they have the same eigenvalues. By this iteration, the matrices A_k converge to an upper triangular matrix, the Schur form of A .

Usually, the QR algorithm starts with the Hessenberg decomposition:

$$H = Q^H A Q \quad (18)$$

where H is a Hessenberg matrix and Q is a unitary matrix. This is because the Hessenberg form is preserved by the QR algorithm and this form can speed up the converge of the QR iteration [6].

There are several ways of Hessenberg reduction such as Gram-Schmidt transformation, Householder reduction and Givens rotation. Efficient parallel algorithm for this Hessenberg reduction is implemented in the ScaLAPACK routine PZGEHRD. So we will focus on the iterative part that comes after this Hessenberg reduction.

The convergence of the Hessenberg QR algorithm can be improved dramatically by introducing spectral shifts into the algorithm. The so-called implicit Q theorem allows us to use the implicit double shift QR algorithm without explicitly computing the QR decomposition. The algorithm is characterized by a "bulge chasing" procedure.

Suppose σ_1 and σ_2 are two shifts of the Hessenberg matrix H . The implicit double shift QR algorithm proceeds as follows:

1. Calculate the first column of the shift polynomial

$$v = (H - \sigma_1 I)(H - \sigma_2 I)e_1 = \begin{pmatrix} * \\ * \\ * \\ 0 \\ \vdots \\ 0 \end{pmatrix} \quad (19)$$

2. Construct a 3×3 Householder transformation Q_1 such that the second and third entries of v are transformed to zero. The similarity transformation gives the updated matrix H_1 :

$$H_1 = Q_1^H H Q_1 = \begin{pmatrix} * & * & * & * & * & * & \cdots \\ X & X & X & * & * & * & \cdots \\ X & X & X & * & * & * & \cdots \\ X & X & X & * & * & * & \cdots \\ 0 & 0 & 0 & * & * & * & \cdots \\ 0 & 0 & 0 & 0 & * & * & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} \quad (20)$$

The Hessenberg structure is damaged by the bulge that we denote with symbol "X".

3. Construct a 3×3 Householder transformation Q_2 such that the third and fourth entries of the first column of H_1 reduce to zero. The similarity transformation gives the updated matrix H_2 :

$$H_2 = Q_2^H H_1 Q_2 = \begin{pmatrix} * & * & * & * & * & * & \cdots \\ * & * & * & * & * & * & \cdots \\ 0 & X & X & X & * & * & \cdots \\ 0 & X & X & X & * & * & \cdots \\ 0 & X & X & X & * & * & \cdots \\ 0 & 0 & 0 & 0 & * & * & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} \quad (21)$$

4. Continue similar operations to chase the bulge. In general, construct a 3×3 Householder transformation Q_k such that the $(k+1)$ th and $(k+2)$ th entries of the $(k-1)$ th column of H_k are mapped to zero. Applying the corresponding similarity transformation to H_k results the updated matrix H_{k+1} , where $k = 2, 3, \dots, n-1$. The bulge will be chased to vanish at the bottom right corner and lead to zeros, thus deflations. For example, H_3 will be look like as follows:

$$H_3 = Q_3^H H_2 Q_3 = \begin{pmatrix} * & * & * & * & * & * & \cdots \\ * & * & * & * & * & * & \cdots \\ 0 & * & * & * & * & * & \cdots \\ 0 & 0 & X & X & X & * & \cdots \\ 0 & 0 & X & X & X & * & \cdots \\ 0 & 0 & X & X & X & * & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} \quad (22)$$

Early shift

For the lossless medium, we have observed that the real eigenvalues of the scattering matrix can be approximated as follows:

$$r_n = \pm \sqrt{\nu^2 - \tilde{\alpha}_n^2} \quad (23)$$

where $-1 < \tilde{\alpha}_n = \sin\theta + n\frac{\lambda}{d} < 1$. We include an example where $M = 15, \nu = 1, \lambda = 1, d = 10.5, \theta = \frac{2\pi}{9}$. The surface used is a generated Gaussian rough surface with correlation length $l = 1$ and standard derivation of height is $rms = 0.2$. The eigenvalues of the scattering matrix are listed in the first column and the second column of Table 1. The eigenvalues in the first column correspond to the outgoing wave and the eigenvalues in the second column correspond to the incoming wave. The values of the $r_n = \pm \sqrt{\nu^2 - \tilde{\alpha}_n^2}$, where $-M < n < M$ are listed in the third column of Table 1:

More experiments show that all the real eigenvalues of the scattering matrix can be approximated by the expression of r_n when the r_n are real. While when r_n has non-zero imagine part, the situation becomes complicated and usually it can not be approximated as in the real case.

Based on this observation, the expression of r_n can be used as shifts to approximate the real eigenvalues of the scattering matrix. The convergence is very quick due to the good approximation. We call this the ‘‘early shift’’. The ‘‘early shift’’ are used in pairs with the sum of the two equal to zero. One pair of the ‘‘early shift’’ will create a bulge to be chased. After the ‘‘early shift’’, all the eigenvalues of the scattering matrix

Table 1: Comparison of eigenvalues and early shift

outgoing waves	incoming waves	$r_n, -M < n < M$
-0.1792 - 1.4278i	-0.1792 + 1.4278i	$\pm 1.8140i$
0.1933 - 1.4471i	0.1933 + 1.4471i	$\pm 1.7044i$
0.1126 - 1.4278i	0.1126 + 1.4278i	$\pm 1.5930i$
-0.1296 - 1.3333i	-0.1296 + 1.3333i	$\pm 1.4794i$
0.0582 - 1.2890i	0.0582 + 1.2890i	$\pm 1.3629i$
-0.0944 - 1.2226i	-0.0944 + 1.2226i	$\pm 1.2428i$
0.0360 - 1.1072i	0.0360 + 1.1072i	$\pm 1.1179i$
0.0113 - 0.9843i	0.0113 + 0.9843i	$\pm 0.9865i$
0.0006 - 0.8504i	0.0006 + 0.8504i	$\pm 0.8454i$
-0.0002 - 0.6893i	-0.0002 + 0.6893i	$\pm 0.6887i$
-0.0000 - 0.5021i	-0.0000 + 0.5021i	$\pm 0.5021i$
0.0000 - 0.2192i	0.0000 + 0.2192i	$\pm 0.2192i$
0.3713 - 0.0000i	-0.3713 + 0.0000i	± 0.3713
0.5529 + 0.0000i	-0.5529 - 0.0000i	± 0.5529
0.6748 - 0.0000i	-0.6748 - 0.0000i	± 0.6748
0.7660 + 0.0000i	-0.7660 - 0.0000i	± 0.7660
0.8368 - 0.0000i	-0.8368 - 0.0000i	± 0.8368
0.8919 - 0.0000i	-0.8919 - 0.0000i	± 0.8919
0.9341 - 0.0000i	-0.9342 - 0.0000i	± 0.9341
0.9652 - 0.0000i	-0.9652 + 0.0000i	± 0.9651
0.9861 + 0.0000i	-0.9862 - 0.0000i	± 0.9860
0.9975 + 0.0000i	-0.9976 - 0.0000i	± 0.9975
0.9996 - 0.0000i	-0.9996 + 0.0000i	± 0.9997
0.9924 + 0.0000i	-0.9922 - 0.0000i	± 0.9929
0.9758 - 0.0000i	-0.9749 + 0.0000i	± 0.9768
0.9478 + 0.0000i	-0.9474 + 0.0000i	± 0.9509
0.9044 - 0.0000i	-0.9094 - 0.0000i	± 0.9144
0.8532 + 0.0000i	-0.8919 - 0.0000i	± 0.8660
0.7613 + 0.0000i	-0.7572 - 0.0000i	± 0.8035
0.6781 - 0.0000i	-0.6757 + 0.0000i	± 0.7233
0.5713 + 0.0000i	-0.5711 + 0.0000i	± 0.6185

are complex, and thus comes in pair and conjugates to each other. The double shift algorithm can be used again and one pair will create a bulge to be chased. Wilkinsons shift can be used after the “early shift”.

So far as we know, this is the first attempt to take advantage of equation (23) to accelerate the convergence of C-method. There do exist other methods other than the C-method that uses equation (23). However these method may have the convergence problem. Interested reader is referred to[4] and [7] for further information.

Parallel bulge chasing

The parallel bulge chasing algorithm was first proposed by Braman et al. [8]. In order to benefit from the level 3 BLAS, they parallelize the bulge chasing procedure by performing the chasing of multiple chains of tightly coupled bulges. With the delay and accumulate technique, the main computation work become the matrix-matrix multiplications. The procedure of intrablock chasing and interblock chasing are described below.

We use the shifts that are mentioned earlier (first “early shift”, then Wilkinson’s shift) to introduce the chain of bulges into the upper left-hand corner of the Hessenberg diagonal sub-matrix that is stored in each processor.

For the intrablock chasing, we may use a sequence of 3×3 Householder transformations to chase the chain of bulges down some rows to the down right-hand corner of the diagonal block. We start from the lowest bulge of the block and chase one bulge at a time. The intrablock chasing can be performed locally and simultaneously which saves computation time. Figure 2 shows how the intrablock chasing are performed, the grey bulges are chased to the black bulges. After the bulge chasing within the diagonal block, the accumulated unitary matrices are sent to the corresponding processors in order to update the off-diagonal blocks. They are then updated by matrix-matrix multiplications which uses level 3 BLAS. The broadcasts are sent in parallel. In order to avoid conflicts in the intersecting parts, they are performed first in the row direction and then in the column direction.

For the interblock chasing, for each diagonal blocks in which the bulge chains reside, we create copys of its neighbors and it becomes similar to the case of intrablock chasing. Figure 3 illustrates the procedure, the grey bulges are chased to the black bulges. More precisely, the processor that stores the grey bulges collects the information of its neighbors and form a copy. Then we can perform the chasing locally, just as in the intrablock chasing. Finally, the updated neighboring block are sent to its owner. To update the outside parts, neighbor processors holding cross-border regions exchange their data in parallel, and the updates are also performed in parallel. This is illustrated in Figure 3 as in the two gray blocks. We perform first the odd-numbered blocks interblock chase, then the even-numbered interblock chase. This odd-even manner avoids conflicts between different tightly coupled chains[9].

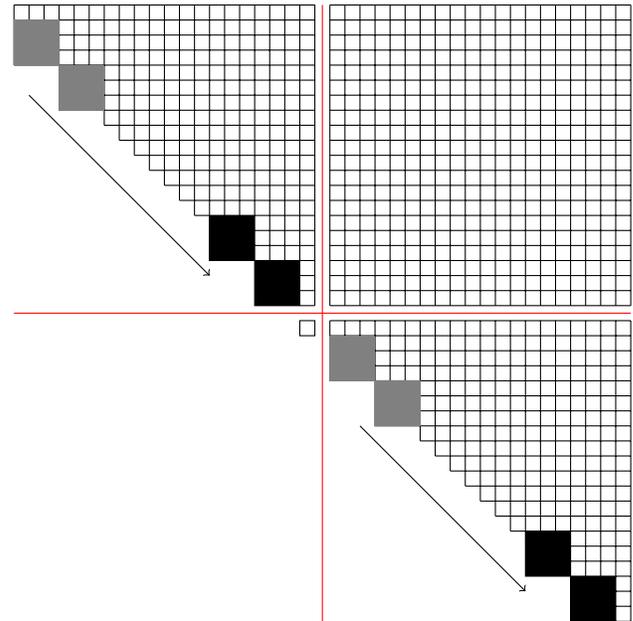


Figure 2: Intrablock parallel bulge chasing

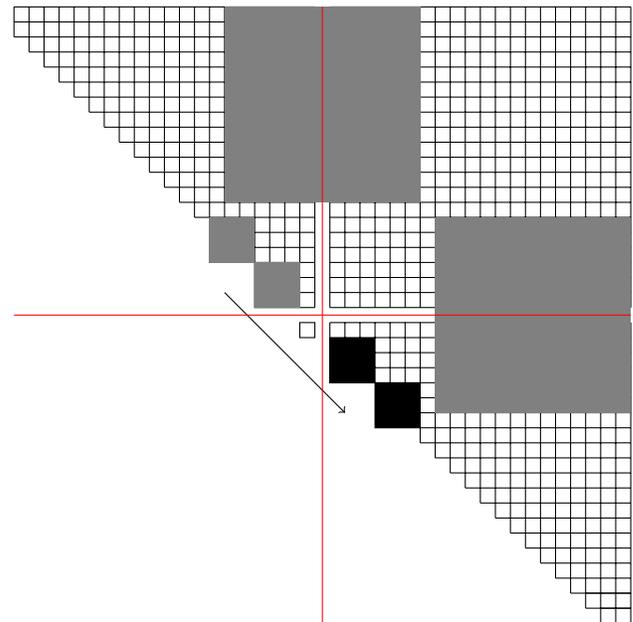


Figure 3: Interblock parallel bulge chasing

The accumulated unitary transformation has the shape as follows:

$$U = \begin{pmatrix} U_{11} & U_{12} \\ U_{21} & U_{22} \end{pmatrix} \quad (24)$$

where U_{12} is a lower triangular matrix, and U_{21} is an upper triangular matrix. So matrix multiplication by U will be broken into two dense by dense matrix multiplications and two triangular by dense matrix multiplications. Computation time is saved because of the triangular structure.

Parallel AED

The parallel aggressive early deflation (AED) algorithm was proposed in [10]. We divide the Hessenberg matrix H as follows:

$$H = \begin{pmatrix} H_{11} & H_{12} & H_{13} \\ H_{21} & H_{22} & H_{23} \\ 0 & H_{32} & H_{33} \end{pmatrix} \quad (25)$$

where H_{11} is of size $(n - k - 1) \times (n - k - 1)$ and H_{33} is of size $k \times k$. We use the pipeline parallel QR algorithm to find the Schur decomposition of H_{33} : $H_{33} = VTV^H$ and perform the following similarity transformation:

$$\begin{pmatrix} I & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & V \end{pmatrix}^H \begin{pmatrix} H_{11} & H_{12} & H_{13} \\ H_{21} & H_{22} & H_{23} \\ 0 & H_{32} & H_{33} \end{pmatrix} \begin{pmatrix} I & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & V \end{pmatrix} \quad (26)$$

$$= \begin{pmatrix} H_{11} & H_{12} & H_{13}V \\ H_{21} & H_{22} & H_{23}V \\ 0 & s & T \end{pmatrix}$$

Now the matrix looks like as in Figure 4. The spike s is denoted as the gray part as in the Figure 4.

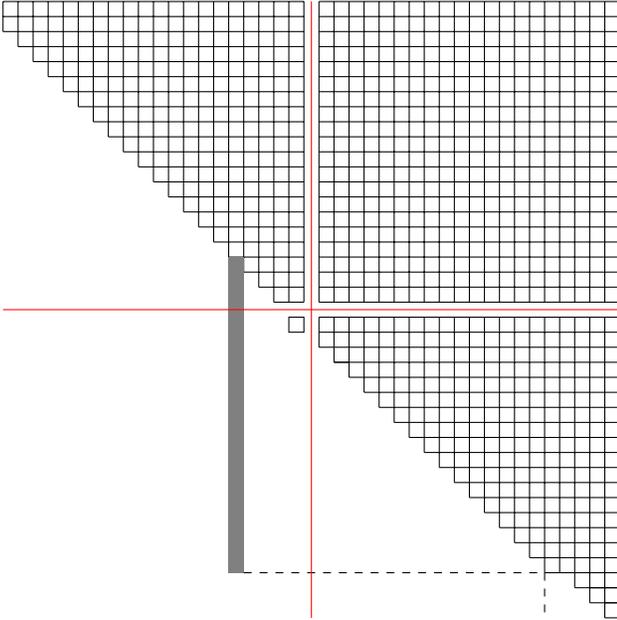


Figure 4: Aggressive early deflation

It has been proved that it is often the case that the some of the last components of s are very small [10]. If it is the case

that the trailing several components of s are negligible, they can set to be zero. The matrix are deflated. This technique often detects convergence much earlier. If it is not the case, we move the eigenvalues to the top left corner of the block H_{33} .

IMPLEMENTATION DETAILS

The experiments are performed on the machine *Poincare*, hosted by Masion de la simulation. This machine is IBM computer, composed mainly iDataPlex dx360 M4 servers:

- 92 nodes equipped with:
 - 2 Sandy Bridge E5-2670 processors (2.60GHz, 8 cores every processor, 16 cores every node)
 - 32 Go memory every node
- 4 nodes GPU equipped with:
 - 2 Sandy Bridge E5-2670 processors
 - 64 Go memory every node
 - 2 Tesla K20 GPU (Cuda Capability 3.5, 4.8 Go memory every GPU)
- 4 interactive front equipped with:
 - 2 Sandy Bridge E5-2670 processors
 - 32 Go memory every node

The following tools are installed: mkl 11.0, intelmpi 4.0.3, lapack 3.5_gnu47, cuda 6.0. The program is written in Fortran and compiled with Intel compiler ifort.

The data layout is as follows:

- The $p = p_r p_c$ processors are arranged into a $p_r \times p_c$ grid. Usually the values of p_r and p_c are set to be as close as possible.
- The $n \times n$ matrix is distributed over the processor grid in a 2D block-cyclic layout. Usually the data blocks are set to be square.

RESULTS

We compare the parallel algorithm with the pipeline parallel algorithm which is implemented in ScaLAPACK routine PZLAHQQR. We also compare the parallel algorithm with or without the “early shift”. In the following experiment, we have the parameters: $\nu = 1, d = 200, \theta = \frac{2\pi}{9}$. The surface used is a generated Gaussian rough surface with correlation length $l = 3\lambda$ and standard derivation of height λ . We set $M = 1000$, so the matrix size is 4002×4002 . In fact, we have performed experiments with different M to see if we have chosen a proper truncation order. If we denote $error = 1 - \sum \epsilon_n$, Figure 5 shows how the function $-\log_{10}(|error|)$ changes with the truncation order M . It shows if we require a precision of 10^{-2} , it should be enough to set $M = 1000$.

Figure 6 shows the comparison between the pipeline parallelization and the parallelization with parallel multishift and AED techniques .

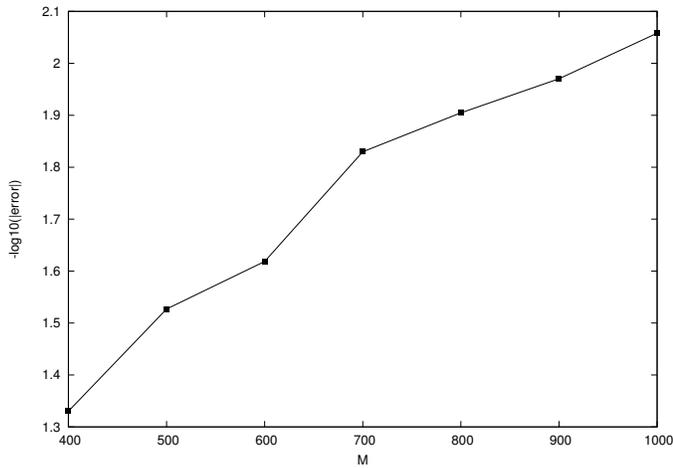


Figure 5: function $-\log_{10}(|error|)$ versus M .

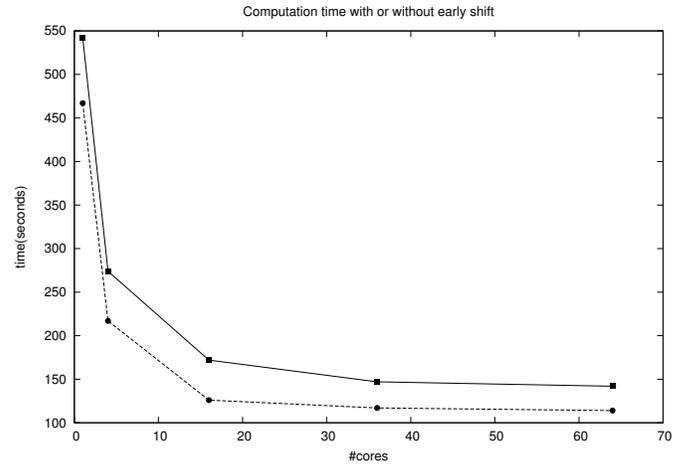


Figure 7: Computation time with or without early shift versus number of cores. The solid line represents the parallel algorithm with parallel multishift and AED, the dashed line represents the parallel algorithm with parallel multishift, AED and early shift

Figure 7 shows that the early shift does speed up the convergence and save the computation time. The performance of PALZHQR with 1 core can be approximated as serial performance. Then, with 64 cores, the computation time of the parallel algorithm with parallel multishift, AED and early shift is approximately $\frac{1}{170}$ of the computation time of serial algorithm.

We also performed an experiment where M and thus the size of the matrix is changing. The number of cores used here is 16 (i.e. 4×4). Figure 8 shows the comparison between the pipeline parallelization and the parallelization with parallel multishift and AED techniques .

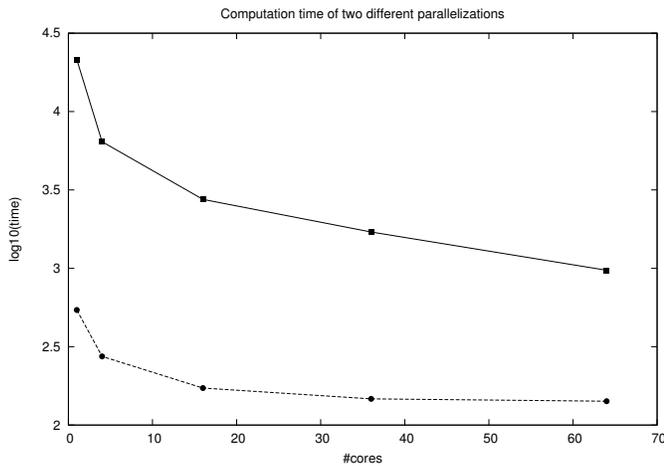


Figure 6: Computation time of two different parallelization versus Number of cores. The solid line represents the pipeline parallel algorithm, the dashed line represents the parallel algorithm with parallel multishift and AED

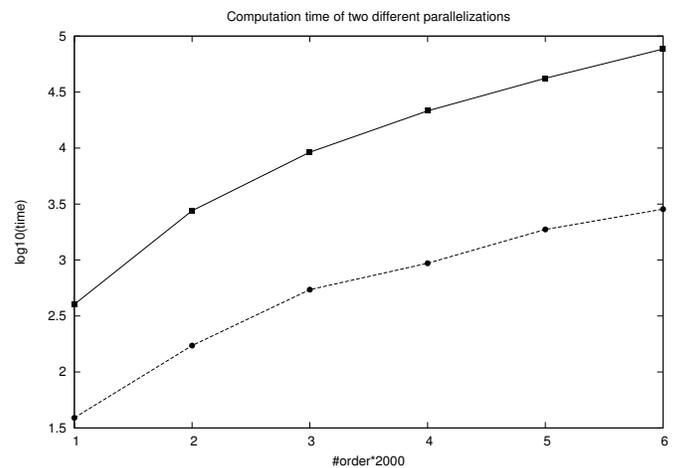


Figure 8: Computation time of two different parallelization versus order of matrix. The solid line represents the pipeline parallel algorithm, the dashed line represents the parallel algorithm with parallel multishift and AED

Figure 9 shows that the early shift does speed up the convergence and save the computation time.

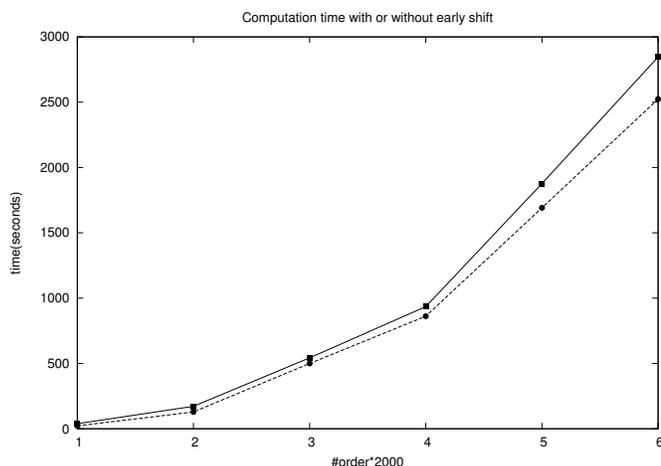


Figure 9: Computation time with or without early shift versus order of matrix. The solid line represents the parallel algorithm with parallel multishift and AED, the dashed line represents the parallel algorithm with parallel multishift, AED and early shift

CONCLUSION

In this paper, we illustrate how to use the C-method to transform the problem of scattering of electromagnetic waves by rough surface to a eigenvalue problem. From abundant numerical experiments, we have observed that the real eigenvalues of the scattering matrix can be approximate efficiently by certain formula. We design the “early shift” algorithm to take advantage of this observation. We plug this “early shift” method, together with Wilkinson’s shift and exceptional shift, into the new parallel QR algorithm. The new QR algorithm uses multiple chains of tightly coupled bulges chasing technique to parallelize the conventional bulge chasing and the aggressive early deflation technique to detect deflation quickly. We apply this specifically designed parallel QR algorithm to our complex, non-symmetric and dense scattering matrix. We also compare the computation time with the pipeline QR algorithm. The results show a significant speed up with our new QR algorithm. This algorithm can be used for analysis of rough surfaces that have very important deformation length. This combination of “early shift” and other shifts can also be used in the problem such as linear-quadratic optimal control problem where a large number of eigenvalues and eigenvectors are needed and background of the original problem can provides very good initial approximations.

REFERENCES

1. J Chandezon, G Raoult, and D Maystre. A new theoretical method for diffraction gratings and its numerical application. *Journal of Optics*, 11(4):235, 1980.
2. Lifeng Li and Jean Chandezon. Improvement of the coordinate transformation method for surface-relief gratings with sharp edges. *J. Opt. Soc. Am. A*, 13(11):2247–2255, Nov 1996.
3. C Baudier, R Dusséaux, K S Edee, and G Granet. Scattering of a plane wave by one-dimensional dielectric random rough surfaces - study with the curvilinear coordinate method. *Waves Random Media*, 14:61–74, 2004.
4. R. Petit. *Electromagnetic Theory of Gratings*. Springer-Verlag. Berlin, Heidelberg, New-York, 1980.
5. Gene Golub and Frank Uhlig. The qr algorithm: 50 years later its genesis by john francis and vera kublanovskaya and subsequent developments. *IMA Journal of Numerical Analysis*, 29(3):467–485, 2009.
6. Gene H. Golub and Charles F. Van Loan. *Matrix Computations (3rd Ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.
7. J.P.Hugonin, R.Petit, and M.Cadilhac. Plane-wave expansions used to describe the feild diffracted by a grating. *J.Opt.Soc.Am*, 71:593–598, 1981.
8. Karen Braman, Ralph Byers, and Roy Mathias. The multishift qr algorithm. part i: Maintaining well-focused shifts and level 3 performance. *SIAM J. Matrix Anal. Appl.*, 23:929–947, 2002.
9. Robert Granat, Bo Kågström, and Daniel Kressner. A novel parallel qr algorithm for hybrid distributed memory hpc systems. *SIAM J. Sci. Comput.*, 32(4):2345–2378, August 2010.
10. Karen Braman, Ralph Byers, and Roy Mathias. The multishift qr algorithm. part ii: Aggressive early deflation, 2002.

A study of manycore shared Memory Architecture as a way to build SOC applications

Yosi Ben Asher	Yousef Shajrawi, Y. Gendel	Gadi Haber	Oren Segal
CS. Dept.	IBM	Intel	EE dep.
University of Haifa	Haifa Labs	Haifa Matam	UMass Lowell
yosi@cs.haifa.ac.il	yousefs/gendel@il.ibm.com	gadi.haber@intel.com	oren@segals.net

November 22, 2014

Abstract

Manycore shared memory architectures hold a significant premise to speed up and simplify SOC's. Using many homogeneous small-cores will allow replacing the hardware accelerators of SOC's by parallel algorithms communicating through shared memory. Currently shared memory is realized by maintaining cache-consistency across the cores, caching all the connected cores to one main memory module. This approach, though used today, is not likely to be scalable enough to support the high number of cores needed for highly parallel SOC's. Therefore we consider a theoretical scheme for shared memory wherein: the shared address space is divided between a set of memory modules; and a communication network allows each core to access every such module in parallel. Load-balancing between the memory modules is obtained by rehashing the memory address-space. We consider practical aspects involved with a practical realization of this scheme, e.g., how will the wire complexity of the communication network affect the execution time.

We have designed a simple generic shared memory architecture, synthesized it to 2, 4, 8, 16, ..., 1024 – cores for FPGA virtex-7 and evaluated it on several parallel programs. The synthesis results and the execution measurements show that, for the FPGA, all problematic aspects of this construction can be resolved. For example, unlike ASICs, the growing com-

plexity of the communication network is absorbed by the FPGA's routing grid and by its routing mechanism. This makes this type of architectures particularly suitable for FPGAs. We used 32-bits modified PACOBLAZE cores and tested different parameters of this architecture verifying its ability to achieve high speedups. The results suggest that re-hashing is not essential and one hash-function suffice (compared to the family of universal hash functions that is needed by the theoretical construction).¹

1 Introduction

The wide use of SOC designs in low power devices creates a demand for high performance SOC's that comprise of multiple cores executing computations in parallel. In this work we propose a scalable 2-1024 cores shared-memory FPGA architecture that can be used to implement such SoCs. Such an architecture will enable to not only speed up computations but also implement the SOC's components in pure software mode.

Typically SoCs contain several processing cycles wherein input signals from sensors are processed and output signals are generated. The units in these processing cycles may be special hardware circuits, con-

¹Extension of Conference Poster FPGA 2014. Almost all of the material in this paper has not been published in FPGA'14 including architecture details, ideas and all the experiments.

SpringSim-HPC, 2015, April 12 - 15, Alexandria, VA, USA

©2015 Society for Modeling & Simulation International (SCS)

trollers, memories, dedicated CPUs and general purpose CPUs. All these units communicate by point-to-point channels or busses and must maintain timing restrictions. Figure 1 (left) illustrates such a processing cycle for a voice recognition system. The different units $U1 \dots U8$ apply signal processing algorithms such as FIR filter, FFT, Viterbi decoding etc. Such a system would include at least one bus several hardware circuits, memories and few cores. A different approach is to implement this voice recognition system as a parallel shared memory program wherein units of the SOC are executed as parallel algorithms (as described in figure 1 right). The communication between the units will be done via shared data-structures such as parallel queues and shared flags. By increasing the amount of cores allocated to each unit, we can increase its processing rates thus allowing the system to maintain desired time constraints. Furthermore, the system is now implemented in pure software mode thus reducing the development effort and simplifying debugging/testing. The goal is thus to replace current SOC designs that combine heterogeneous cores and hardware accelerators by a set of many homogeneous light cores communicating through shared memory. Note that the proposed architecture implements a close model to the well known PRAM model for parallel computations allowing the use of the vast set of PRAM algorithms that has been developed.²

In order for such a shared memory architecture to be used in a SOC it must satisfy the following requirements:

- Support large number of cores in a single chip/FPGA so that each unit of a processing cycle can be sped up to meet necessary time constraints.
- The clock latencies should be about the same as current SoCs.
- It should not consume extra power compared to current SoCs.
- Shared memory accesses must be made in very few clock cycles since the shared memory is used

²The term “close” is due to the fact that memory references may suffer unexpected delays thus the proposed architecture only approximates the synchronous steps of the classic PRAM.

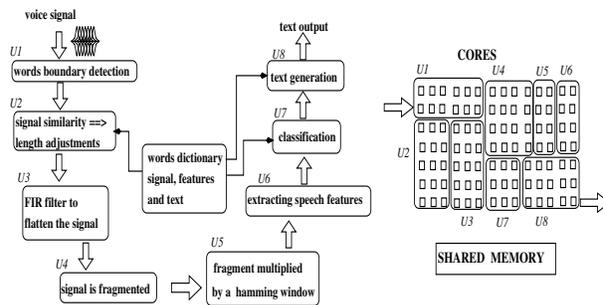


Figure 1: Processing cycle of Voice recognition and its embedding in a shared memory multicore architecture.

for both the communication between the SoC’s units and the parallel processing inside each unit.

However, current shared memory multicore chips support only a relatively small number of cores since shared memory is implemented via cache coherency protocols [14]. This form of shared memory simulation is not scalable and is subject to large delays needed to propagate updated consistent values of shared variables between the caches and the main memory. A more direct way to build shared memory called DMM (Distributed Memory Machine) was proposed as a theoretical implementation for the PRAM model [8]. The DMM uses a multistage network to transfer read/write packets to a set of memory modules each holding a distinct part of the shared address space. This suggests an alternative way to build very large manycore chips where a DMM shared memory replaces the use of cache-based shared memory. Although the theoretical construction of PRAM is well defined, it is not clear how and if at all it will practically function for a high number of cores in a SoC mode. Thus in this work, we study the usefulness of the DMM solution for building 2-1024 manycore FPGA architecture which, as indicated before, can form an alternative architecture for SOCs.

It is important to understand how, theoretically, shared memory of a multicore machine can work. In the proposed theoretical construction, the shared address space is partitioned to n memory modules. Each core can access any module via a multistage communication network (such as a butterfly network

(BN) or a hypercube). Every global address can be resolved to a module + internal address in that module. Routing the memory requests to the suitable module and back is done through a communication network. The memory space is hashed so that the probability of n load/store instructions executed by n cores to issue more than $\log n$ memory addresses that have been mapped to the same memory module is polynomially small. Thus the fact that each memory module can serve only one core at a time is not likely to delay the execution of these n instructions significantly. This hashing of memory addresses is achieved by using a hash function $h(x)$ pre-selected randomly from a finite family of universal hashing functions $H = \{h_1(x), \dots, h_k(x)\}$ [4]. Each memory reference to an address x is directed (by all cores) to a different address $h(x)$. Intuitively a random selection of $h(x) \in H$ before the program begins, guarantees that the probability of directing more than $\log n$ shared-memory references onto the same module, is polynomially small in the number of cores/memory-modules n . This result is further extended for $\log n$ -wise independent hashing functions guaranteeing that the queue size at a given module is no greater than $\log n$. If the queue size at a memory module is found to be too small during execution, then a different hash function is chosen and the memory is rehashed. This can take some time but the probability of rehashing is low and decreases asymptotically as the length of the program increases. Implementing this theoretical scheme fully may lead to a non-practical design. Thus in this work we study the following main issues:

- How is the memory latency affected by increasing n (number of cores/memory-modules). Practically we need this latency to remain about the same even when the number of cores increases.
- Does the construction work for small values of n or is it valid only for large values of n (e.g., $n = 512, 1024$)?
- How frequent if at all, do we really need to use the memory re-hashing mechanism? Can a small and simple enough family H be used?
- What is the expected latency of a memory reference due to multiple memory references that are

directed to the same memory module? We hope to show that this delay is very small.

- What is the effect of collisions of packets in the BN-switches (congestion)? Since the use of universal hashing is similar to using random destinations, we hope to show that the penalty of packets congestion in the BN is limited. Though theoretical solutions to eliminate BN collisions do exist, they are too complicated to be realized. Thus it is important to see that practically there is no need to use them. In this case we can eliminate the need to use buffer-queues in the BN-switches and even run the BN as a pure combinatorial circuit.
- The BN requires increased wire area of n^2 which can also affect the execution frequency. We hope to show that this is not the case and the internal routing mechanism of the FPGA absorbs this complexity.

A positive answer to these questions implies that the theoretical DMM can be used in a very simple and basic mode for SOCs over FPGAs.

Implementing SOCs as many cores working in parallel can also help reduce power consumption. Power is proportional to *voltage*² (V^2), hence it can be significantly reduced if we are able to reduce the voltage V . However reducing V requires a lower execution frequency. In a many core system, the performance loss caused by the lower execution frequencies can be compensated by the increased parallelization. For example, assume that a given task executed by one fast-core at a given voltage V and frequency f is split to k cores. Ideally, it is now possible to let each of these k cores run at voltage V/k and at frequency f/k still maintaining the original execution time of the fast-core. However, due to the quadratic effect of lowering the voltage, the power consumption of these k cores is $1/k$ of the power consumption of the fast-core. Thus using 2-1024 cores at a slower frequencies may lead to reduced power consumption.

2 Proposed architecture

In this work we study a practical version of this theoretical construction. This architecture does not use

caches, rather relying on the ability to route memory requests to the suitable memory modules in one or few clock cycles. On the other hand, we use a butterfly network (BN) whose complexity may prevent the low memory latencies that are needed for a practical use of this type of architecture. In particular, the 2D layout of the BN requires an order of n^2 area of wires to connect between its $n \cdot \log n$ switches thus potentially incur long delays. One option is to realize the BN as a multi-stage network wherein packets are stored in the switches and advance to the next level every clock cycle. Another possibility is to implement the BN as a pure combinatorial circuit. The use of multi-stage network where in packets advance to the next level every clock cycle may potentially lead to small latencies since the clock frequency counts only for the latency of one switch. However, it will take $\log n$ steps to route packets through the BN which violates our need for one (or very-few) clock cycle memory latencies. The other option of using a pure combinatorial realization of the BN seems like a better option. In particular, there is a possibility that the complexity of the BN will be absorbed by the FPGA reconfigurable infrastructure and thus the routing complexity of the BN will not affect the memory latencies. This is because FPGAs have already a network which is a crossbar of buses that is used to dynamically interconnect between the different hardware components (such LUTs, registers,...) of the FPGA. Thus, the FPGA may be able to embed the BN without incurring increased latencies. We have built such a multi-core architecture and verified that the FPGA's infrastructure can indeed absorb the complexity of the BN.

The proposed multicore contains the following components (as depicted in figure 2):

- Programming mode is SPMD where each core executes one thread that is loaded and executed from a pre initialized memory module that is local to the core. Though this is a simplified mode of programming it is sufficient to demonstrating the usefulness of such multicores to embedded systems. A simple compiler for C + ASM is used to automatically compile SPMD C programs to n cores.

- A set of $n = 2^k$ $k = 0, \dots, 10$ memory modules M_1, \dots, M_n form the memory shared address space of n Pacoblaze cores (A clone of Xilinx Picoblaze). These are light FPGA soft CPUs that have been modified to support 32 bits (values and memory addresses) and fixed point arithmetics. Each core has call/return stack and an instruction ROM that are managed through local memory modules.
- Memory references are generated as packets where each packet contains the following fields:

[*R/W - bit* | *memory - module* | *internal - address* | *priority*]

- A multistage butterfly topology network (BN) connecting the n memory ports of the cores to the shared memory modules. The BN works as a combinatorial circuit allowing packets to be routed to their destination in one clock cycle (unless collided with another packet). The use of a BN as a combinatorial circuit is thus done in order to save wire area compare to the ability to use a circuit that selects for every memory module the next packet that is destined to it. Thus the whole point of using a combinatorial BN-circuit is to reduce the wire/MUX complexity if we directly connect each core to each memory module. Figure 3 illustrates collision of two packets attempting to use the same exit of a switch where one sent from core 0010 to module 1010 and the other from 1011 to 1000.

When a packet is dropped at a given switch the core that issued this memory reference is blocked and the memory request is repeated in the next clock cycle until the packet reaches its destination. Note that the failing signal is not propagated directly to the core but through the backward BN's switches. Thus the core simply waits until the acknowledge-packet is received at the suitable port of the backward BN. We thus skip the optimization not to wait for the completion of store operations as this would prevent sequential consistency [3]. For example, If two cores execute the program

core-1 : $x = 2; y = 1;$ *core-2* : $while(y == 0); print(x);$

and the initial values are $x = y = 0$ then under sequential consistency the program can not print $x = 0$ which can happen if core-1 will not wait for write-ack.

- When a packet pa is dropped in a switch $S_{i,j}$ it is stored in $S_{i,j}$ in an intermediate-register and will continue in the next clock cycle, not from the original core that issued it, but from $S_{i,j}$. This complicates the logic at each switch but, as depicted in figure 3, can: a) Make free the switches in the path of pa to $S_{i,j}$ so that they can be used by other packets (e.g., pes in the figure). or b) Prevent a possible “left blocking” of pa at the next clock cycle by a packet pc with a higher priority. Collisions (as depicted in figure 3) can now occur not only between two packets entering the BN’s but also with the packet that resides in the intermediate register. In this case (three packets collide) one of the packets is dropped back to the core that issued it, one stays in the intermediate register and one continues to the next level.
- The decision which packet is dropped and which continues is made based on a “priority” of the packets arriving to the switch. We considered the following possibilities:
 - (1)- No priority, e.g., drop the packet sent from the left entry of the switch.
 - (2)- Prefer packets that where sent from smaller core-ids;
 - (3)- Prefer older packets (using time tags);
 - (4)- A packet starts with priority=0 and its priority increases every time it collide and is dropped.
 - (5)- Combination of (4)- i (3)- i (2) by this order.

We selected option (4) which is the most efficient in terms of latency.

- The theoretical shared memory uses universal hashing scheme [13] to re-map memory addresses so that, with high probability, the number of parallel load/store accesses destined to the same memory module is less than $\log n$. The theoretical framework requires that one hash-function should

be selected in random from a family of n hash functions before a program is executed. This is not practical, we have tested a smaller family of such functions that can be realized without increasing the clock latency or add extra clock cycles to load/store operations.

- Each Pacoblaze core has an I/O-port however these ports can not be left as external ports due to the large number of cores. The cores’ I/O-ports are connected through a bus to an external I/O device (e.g., Ethernet card, Camera or a cellular modem) that can broadcast/collect packets of data to/from local I/O buffers of each core ($I/O - B_0, \dots I/O - B_7$ in figure 2). Each core has a controller that collects the data items of a data-packet that is sent to it by the I/O device or outputs data-items to the external I/O device. The fact that I/O is handled through a local memory at each core, reduces the amount of shared memory transactions needed otherwise. Note that the external I/O device should be able to direct data-packets to specific cores. A control line is used to indicate that the core is not ready to accept another packet and that the I/O device should send it to another core. The I/O device can use the control to indicate that the bus is not free to write packets. Note that it is possible to ignore these control signal and let the I/O device transfer packets at a fixed rate assuming that $\#cores \cdot local_processing_time$ is sufficiently large to allow it to send packets to cores in a fix-rate. Thus the rate in which data/packets can be injected into the chip scales-up with the number of cores which is an important factor for SOC applications that need to process a stream of incoming data in high rates.

Figure 2 illustrates the proposed architecture for $n = 8$ cores ($C_0 \dots C_7$), showing the BN (forward and backward), partition of the shared address space to memory modules, the hashing units, the ring-buffer with two real ports and the interrupt-buss.

It is important to note that the packet lost at the BN does not hamper sequential consistency since the three conditions of sequential consistency [3] are trivially preserved:

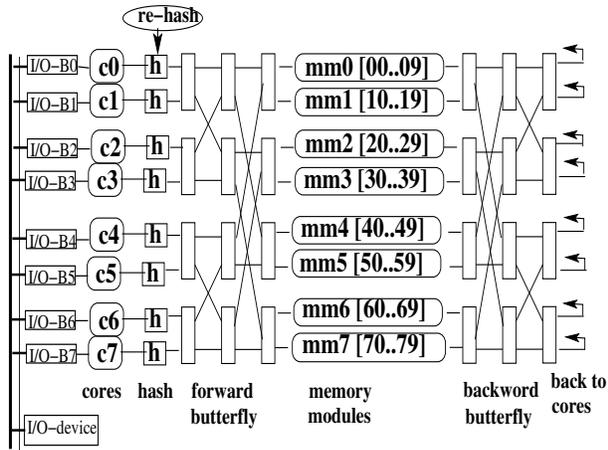


Figure 2: Schematic layout the PRAM multicore architecture.

- Cores issue memory operations in program order as the Pacoblaze cores are not out-of-order and so is the compiler.
- Each core waits (through the backward BN) for every store operation it has started to complete before issuing another operation.
- A load operation L by $core_i$ that returns a value from of a store S performed by $core_j$ can not terminate before $core_j$'s store terminated. This holds because the acknowledge/return-value of L will arrive at-least one clock cycle after the acknowledge of S will arrive to $core_j$. Note that the use of intermediate registers at the switches does not damage the sequential consistency.

It trivially follows that there is no starvation and the priority mechanism that was implemented creates a fair execution since packets that collide and are dropped (back to the core or to an intermediate register) acquire a higher priority.

3 Experimental results

First we show the synthesis results of the proposed architecture for the FPGA using Xilinx's ISE 13.1. The results of the FPGA synthesis of this architecture are depicted in table 1 showing that the clock latency

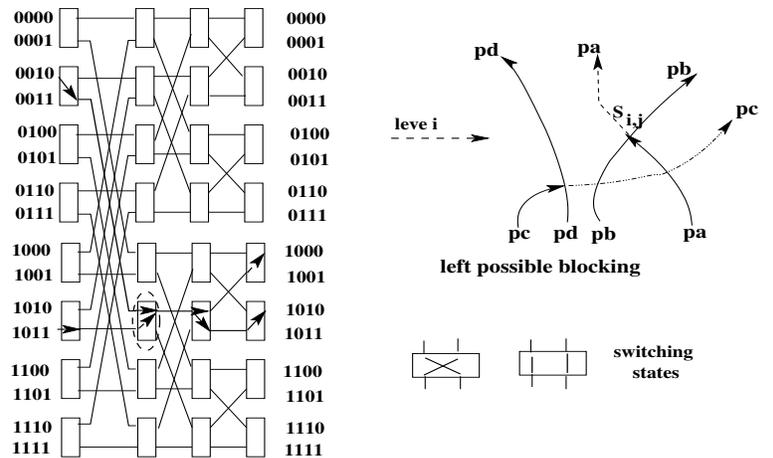


Figure 3: Butterfly network+collision and avoiding collisions due to intermediate restarting of packets.

does not increase significantly when more cores are used. The fact that the resources (LUTs+Registers) at most doubles every time we double the number of cores prove that the connections between the BN-switches is absorbed by the FPGA. It thus follows that the BN can be embedded by the routing infrastructure of the FPGA without increasing the memory latency. Though there is a small decrease in the clock frequency when more cores are used it is very slow compare to the double increase in the number of cores. Clearly resources and power grow proportionally in the number of cores. $n = 64$.

cores	Slice LUTs	Slice Registers	DSP	Dynamic power (W)	Logic power (W)	freq. Mhz
1	872	118	4	0.04	0.004	101
2	1854	237	8	0.082	0.009	101
4	4350	482	16	0.168	0.02	99
8	8621	979	32	0.34	0.042	98
16	16785	1988	64	0.681	0.088	96
32	37274	4037	128	1.416	0.193	95
64	81216	8198	256	2.92	0.418	95
128	171731	16647	512	5.964	0.901	95
256	363975	33800	1024	12.189	1.903	87
512	756602	68617	2048	25.003	4.069	86
1024	1813814	139274	3568	54.286	9.941	83

Table 1: Synthesis results for the FPGA Virtex-7 and ASIC.

Next we show the speedup and effect of collisions (at the memory modules and at the BN's switches) for several parallel programs. These results are given for the Reverse-LSB hashing function that achieved maximal speedups for all programs. A more detailed results showing the effect of using different hash-

function will be given next for the Viterbi decoding algorithm. The results Table 2 show high speedups and average of 1-2 clock cycle per memory reference.

program	intermediate register	min load	max load	cycles / conflicts	speedup
odd-even merge sort	no	304225	648205	6.84	480.60X
Matrix multiplication	no	2445329	2457127	443.06	1019.08X
sum	no	9384	473474	1.18	20.30X
N-body	no	1406618	1815890	14.59	547.70X
Prime numbers	no	64519	87931	36.28	751.36X
List ranking	no	74554	852179	1.02	1024.00X
Viterbi	no	239957	308311	10.85	796.97X
Bloom filter	no	29895	61645	5.11	496.59X

Table 2: Speedup/collisions for parallel Reverse-LSB of several parallel programs (1024 cores).

Table 3 shows the results of running a parallel implementation of the viterbi algorithm using a 1024-core configuration. This algorithm is used in decoding the convolutional codes used in both CDMA and GSM digital cellular networks, in addition to various other types of wireless networks. The results indicate that the Reverse LSB function gave the best speedup, 796.97X without the intermediate register and 818.21X with the register while maintaining a good conflict rate of 1 conflict per 10.85 and 12.18 cycles respectively. The Reverse MSB function obtained the worst results both speedup wise and cycles/conflict wise: 118.94X and 1.25 respectively with the intermediate register disabled. Recall that in Viterbi algorithm we have an automata A with probabilities on its state transitions (called inner transitions) and probabilities of observing a set of output symbols from each state. The goal is, given a sequence of observable outputs $X = x_1, \dots, x_k$ to find the most likely sequence of inner transitions that produced the given sequence of observable outputs (i.e., the sequence with the highest accumulating probabilities). In order to compute this sequence we build a “trellis” of all possible state-transitions where each level i in the trellis contains the accumulate probabilities of all possible inner transitions that can produce the prefix x_1, \dots, x_i . Note that in this experiment we used a large hidden automata with 1024 states, thus there where 10^6 transitions between consecutive levels of the trellis and hence a large number of computations that can be done in parallel.

Finally we evaluated (table 4) the effect of the priority mechanisms described in section 2) for prime

Hash-func	intermediate register	min load	max load	cycles / conflicts	speedup
LSB	no	239959	426420	4.51	576.23X
Reverse LSB	no	239957	308311	10.85	796.97X
MSB	no	239957	2002021	1.28	122.70X
Reverse MSB	no	239957	2065937	1.25	118.94X
Middle Bits	no	239957	1180715	1.54	211.04X
Reverse Middle Bits	no	239957	1151426	1.54	213.40X
Circular Rotate Left	no	240020	1038006	1.68	236.72X
Reverse Circular Rotate Left	no	240020	957105	1.78	256.73X
Left					
LSB	yes	239959	378979	5.43	648.36X
Reverse LSB	yes	239956	300307	12.18	818.21X
MSB	yes	239957	2025312	1.25	121.32X
Reverse MSB	yes	239957	2025312	1.25	121.32X
Middle Bits	yes	239957	1164324	1.54	211.04X
Reverse Middle Bits	yes	239957	1152715	1.54	213.16X
Circular Rotate Left	yes	240020	1013257	1.71	242.50X
Reverse Circular Rotate Left	yes	240020	955576	1.78	257.14X

Table 3: Speedup/collisions for Viterbi algorithm. 1024cores

numbers, sorting and nbody (using the Reverse LSB hash-function). Note that all previous results were obtained with the no-priority scheme (1). All of the priority mechanisms improved the speedups e.g., a speedup of 1002.19X for primes using core-ids compare to 751.36X with no priority. The sorting program used barrier locks between iterations for which the core-ids scheme caused starvation (the ----- result in table 4). The nbody measurements were applied to the program after all locks have been removed so that the core-id scheme will not cause starvation. Basically these results suggest that using the number of drops priority is a good choice since it is simple to implement, starvation-free and obtained good results.

priority mechanism	speedup primes	speedup sorting	speedup nbody
1 (none)	751.36X	480.60X	602.59X
2 (code id)	1002.19X	-----	582.82X
3 (time stamp)	993.09X	560.27X	667.49X
4 (number of drops)	993.09X	560.27X	667.49X
5 (combo of 2-4)	991.68X	559.93X	668.12X

Table 4: Effect of different priority mechanisms on prime numbers and sorting.

In conclusion, the above experiments indicate that shared memory 1K-core chips can be implemented in the proposed manner as follows:

- Most of the algorithms gave good speedups and good scalability.
- It seems that two hashing functions (LSB and Reverse LSB) suffice for all benchmarks. This because most algorithms traverses arrays/list ar-

ranged linearly in the memory the Reverse LSB simply distributed these data-structures evenly between the memory modules such that consecutive addresses were now mapped to different memory modules. This reduced the amount of collisions at the BN switches and at the memory modules to a minimum.

- The experiments assert the usefulness of the intermediate register technique which is fairly simple and do preserve the one cycle latencies of the BN (the BN basically remains a pure combinatorial circuit). We remark that we have implemented a BN with waiting buffer-queues at every node. However, using such a BN increased the amounts of resources and latencies significantly up to a point where large number of cores could not be used.
- Seems that routing complexity of the BN has no significant effect for the FPGA realization making attractive for FPGA realizations.

4 Related works

The theoretical construction of DMM to simulate PRAM was studied in several works: [15] proposed a PRAM DMM architecture that use universal hashing not only to eliminate congestions at the memory modules but also eliminate congestion at the communication network that connects the cores to [8] the memory module. This method used a complex scheme imposing a monotone order for transferring memory references between the switches of the communication network. The SB-PRAM [9] is an attempt to implement Ranade's scheme in real hardware. However, as indicated, we hope to show that congestion in the communication network will practically not affect the performance even when Ranade's scheme is not used.

The idea of building a PRAM (Parallel Random Access Model) on a single chip, where the chip include the cores, memory-modules and the communication network) was proposed and to some extent implemented in [1] by U. Vishkin [16]. Different types of CPU-to-Memories interconnection networks have

been studied and compared. The architectures presented do not include resource-sharing bus, which is similar to the architecture we choose to implement. [16] proposes an FPGA-based system and exploits the explicit Multiple-Threading programming model. Next [5] proposed a hardware architecture suitable for performing one of the main tasks of 21-st century's challenge of information society, namely the system able to perform parallel computing efficiently. This architecture uses a collection of symmetric (non-dedicated) processors working with the distributed memory blocks (DMM) while the memory is shared from the programmer's point of view. One of the goals of this architecture is to simplify the programming model by using a very complex hardware Scheduler Unit, while we propose simpler hardware solution by off-loading some software engineering tasks to the compiler implementing SMT and using the dedicated ISA for this purpose.

Unfortunately, by definition, the bandwidth of a BN is restricted. Shared-memory cannot be simulated on a BN while still retaining optimal efficiency (optimal efficiency is where the time-processor product is within a constant ratio of the sequential version). Although the latency from the routing delay across the network can be hidden with parallel slackness (multiple PRAM processors are simulated on one physical simulating processor), network congestion delays cannot. If requests are initiated at times 0 and 1 then a network delay of L will lead to replies in L and $L+1$ but if the delay L is caused by congestion then the replies will come in L and $2L$ time. However while memory requests are limited in number contention can be restricted using hashing.

In turning our proposed SoC into ASIC, there's a requirement for large amounts of on-chip memory. this is achievable with a three-dimensional integrated circuit [2] combining the PRAM in a 3D-stacked memory fashion with a many-core design [6]. Current developments in the field of memristors promise memristor arrays built on a CMOS chip [10], while others showed the benefits to bandwidth and latency of stacking DRAM on-die [7], as such, we strive toward an ASIC design combining a many-core architecture with on-die PRAM.

It is also important to note that a drive towards

implementing, in software, parts of the SoC that are traditionally done in hardware, such as a modem or a radio, and running them on a custom multi-core chip is something already being done in latest commercial chips such as the case in NVIDIA's software defined radio technology [12]. Unlike the current NVIDIA implementation, which uses 8 cores and lacks a huge on-chip shared memory. our proposed chip provides a much larger amount of (simpler) cores connected to a shared memory system that can be up to few gigabytes in our current 32-bit implementation

It is important to compare PRAM on a chip to GPGPU architectures and in particular to Fermi [11] which contains 512 "cores" supporting a small global shared memory. Unlike PRAM on a chip or Multicores, GPGPUs works best in single instruction multiple data (SIMD) mode supporting efficient execution of scatter/gather load/store instructions. This is because only if data addresses produced by the GPU's cores fall in the same DRAM/CACHE row/line then low memory latencies can be obtained. This also contains high entry overhead from the end-programmer and requires following complicated rules. In PRAM on a chip or multicore there is a uniform cost of all memory references made by the different cores even for arbitrary addresses and consequently programming style supports general threads. [?] contains a short indication to a shared memory manycore architecture, however the architecture itself and the experimental results have not been described.

References

- [1] P. Bach, M. Braun, A. Formella, J. Friedrich, T. Grun, and C. Lichtenau. Building the 4 processor sb-pram prototype. In *HICSS '97: Proceedings of the 30th Hawaii International Conference on System Sciences*, page 14, 1997.
- [2] B. Black, M. Annavaram, N. Brekelbaum, J. DeVale, L. Jiang, G. H. Loh, D. McCaule, P. Morrow, D. W. Nelson, D. Pantuso, et al. Die stacking (3d) microarchitecture. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pages 469–479. IEEE, 2006.
- [3] D. E. Culler, A. Gupta, and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., 1st edition, 1997.
- [4] M. Dietzfelbinger and F. Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. *Automata, Languages and Programming*, pages 6–19, 1990.
- [5] R. Ginosar and N. Bayer. Tightly coupled multiprocessing: Architectural and technical challenges.
- [6] K. D. Hyun et al. 3d-maps: 3d massively parallel processor with stacked memory. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International*, pages 188–190. IEEE, 2012.
- [7] D. Jevdjic, S. Volos, and B. Falsafi. Die-stacked dram caches for servers: hit ratio, latency, or bandwidth? have it all with footprint cache. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.
- [8] R. Karp, M. Luby, and F. Meyer auf der Heide. Efficient pram simulation on a distributed memory machine. *Algorithmica*, 16(4):517–542, 1996.
- [9] J. Keller, W. Paul, and D. Scheerer. Realization of prams: Processor design. *Distributed Algorithms*, pages 17–27, 1994.
- [10] K. Kuk-Hwan et al. A functional hybrid memristor crossbar-array/cmos system for data storage and neuromorphic applications. *Nano letters*, 12(1):389–395, 2011.
- [11] J. Nickolls and W. Dally. The gpu computing era. *Micro, IEEE*, 30(2):56–69, 2010.
- [12] NVIDIA. Nvidia software defined radio modem technology, the modem innovation inside nvidia i500 and tegra 4i. *NVIDIA Whitepaper*, 2013.

- [13] A. Ostlin and R. Pagh. Uniform hashing in constant time and linear space. In *STOC '03: Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 622–628. ACM, 2003.
- [14] M. S. Papamarcos and J. H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. *SIGARCH Comput. Archit. News*, 12:348–354, 1984.
- [15] A. Ranade. How to emulate shared memory. *Journal of Computer and System Sciences*, 42(3):307–326, 1991.
- [16] X. Wen and U. Vishkin. Fpga-based prototype of a pram-on-chip processor. In *CF '08: Proceedings of the 5th conference on Computing frontiers*, pages 55–66, 2008.

Solving the Klein-Gordon equation using Fourier spectral methods: A benchmark test for computer performance

S. Aseeri¹
samar.aseeri@kaust.edu.sa

O. Batrašev²
olegus@ut.ee

M. Icardi³
matteo.icardi@kaust.edu.sa

B. Leu⁴
brianleu@umich.edu

A. Liu⁴
alberliu@umich.edu

N. Li⁵
ning.li@nag.co.uk

B.K. Muite^{2,6}
benenson.muite@ut.ee

E. Müller⁷
e.mueller@bath.ac.uk

B. Palen⁸
brockp@umich.edu

M. Quell⁹
michael.quell@yahoo.de

H. Servat¹⁰
harald.servat@bsc.es

P. Sheth¹¹
pssheth@umich.edu

R. Speck¹²
r.speck@fz-juelich.de

M. Van Moer¹³
mvanmoer@illinois.edu

J. Vienne¹⁴
viennej@tacc.utexas.edu

ABSTRACT

The cubic Klein-Gordon equation is a simple but non-trivial partial differential equation whose numerical solution has the main building blocks required for the solution of many other partial differential equations. In this study, the library 2DECOMP&FFT is used in a Fourier spectral scheme to solve the Klein-Gordon equation and strong scaling of the code is examined on thirteen different machines for a problem size of 512^3 . The results are useful in assessing likely performance of other parallel fast Fourier transform based programs for solving partial differential equations. The problem is chosen to be large enough to solve on a workstation, yet also of interest to solve quickly on a supercomputer, in particular for parametric studies. Unlike the Linpack benchmark, a high ranking will not be obtained by simply building a bigger computer.

Author Keywords

Parallel Algorithms; Fast Fourier Transforms; Benchmarks; Partial Differential Equations;

ACM Classification Keywords

D.1.3 Concurrent Programming: Parallel programming; F.2.1 Numerical Algorithms and Problems: Computation of transforms; C.4 Performance of Systems: Measurement tech-

niques; G.1.8 Partial Differential Equations: Spectral methods

1. INTRODUCTION

The focusing nonlinear Klein-Gordon equation describes the evolution of a possibly complex scalar field u according to

$$u_{tt} - \Delta u + u = |u|^2 u, \quad (1)$$

where t is time and $\Delta = \partial_{xx} + \partial_{yy} + \partial_{zz}$ the three-dimensional Laplacian. In this note we will focus on real valued functions u only although the programs used allow u

¹KAUST Supercomputing Laboratory, 4700 King Abdullah University of Science and Technology, Thuwal 23955-6900, Kingdom of Saudi Arabia, ²Arvutiteaduse instituut, Tartu Ülikool, Liivi 2, Tartu 50409, Estonia, ³Stochastic Numerics Group, 4700 King Abdullah University of Science and Technology, Thuwal 23955-6900, Kingdom of Saudi Arabia, ⁴Electrical and Computer Engineering, University of Michigan, Ann Arbor, MI 48109-2122, USA, ⁵Numerical Algorithms Group, Wilkinson House, Jordan Hill Road, Oxford, OX2 8DR, United Kingdom, ⁶المملكة العربية السعودية ٢٣٩٥٥-٦٩٠٠

٢٠١٥-٢٠١٦ جامعة الملك عبد الله للعلوم والتقنية، ثول ٢٣٩٥٥-٦٩٠٠، مجموعة الرياضيات العددية (Numerical Mathematics Group, 4700 King Abdullah University of Science and Technology, Thuwal 23955-6900, Kingdom of Saudi Arabia), ⁷Department of Mathematical Sciences, University of Bath, Bath BA2 7AY, United Kingdom, ⁸CAEN Advanced Computing, University of Michigan, Ann Arbor, MI 48109-2094, USA, ⁹Institut für Analysis und Scientific Computing, Technische Universität Wien, Wiedner Hauptstrasse 8-10, 1040 Wien, Austria, ¹⁰Computer Sciences Department, BSC, Department of Computer Architecture, UPC, c/Jordi Girona, 31 - 08034 - Barcelona, Catalunya, Spain, ¹¹Biomedical Engineering, University of Michigan, Ann Arbor, MI 48109-2122, USA, ¹²Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH, Germany, ¹³National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA, ¹⁴Texas Advanced Computing Center, University of Texas at Austin, Austin, Texas 78758-4497, USA.

to be complex. This equation can exhibit a phenomena known as blow up in which the field u becomes infinite at a certain point. Many partial differential equations which model physical phenomena can exhibit blow up, which typically indicates that the differential equation model is no longer a good model for the physical phenomenon under investigation. Understanding under which initial conditions a blow up in the Klein-Gordon equation will occur is still not clear and it is hoped that simulations can help to elucidate this. Some parametric numerical studies of three dimensional radially symmetric real solutions to the Klein-Gordon equation have recently been done by Donninger and Schlag [37]. Their parametric study generated a large amount of data, which was unfeasible to store. Their study also made an assumption of spherical symmetry to be able to solve a one-dimensional equation for the radial component of a three-dimensional field. It is of interest to perform similar parametric simulations in three dimensions without symmetry assumptions for the Klein-Gordon and other partial differential equations. Fourier spectral methods are an effective tool for doing this on petascale supercomputers.

The purpose of this paper is to conduct strong scaling experiments of a full model problem on different platforms, trying to understand how much faster a moderate size problem can be made to run. The Klein-Gordon equation is a simple model application for which different choices of numerical methods and computer architectures can be tried to determine which one will give an accurate enough solution at either the fastest time, lowest computational resource or lowest energy cost.

To limit the scope of this study, parallel fast Fourier transforms (FFT) are done with the library 2DECOMP&FFT [48, 1] which primarily uses `MPI_ALL_TO_ALLV` or `MPI_ALL_TO_ALL` for its communications. However, it would be interesting to perform a similar study with other parallel Fourier transform libraries (for example PFFT [56, 14], OpenFFT [38, 12], PKUFFT [31, 15] and P3DFFT [55, 13]).

The focus of the study is on a discretization of 512^3 since (i) there are still a wide variety of numerical experiments that can be done for problems of this size, (ii) post processing can be done locally on workstations and (iii) for parametric studies, it is unclear whether access to high-performance computers or a distributed computing cloud based solution is most appropriate. It would also be interesting to try out other numerical methods, such as higher-order time stepping, a lattice-Boltzmann method [49] and an implicit finite difference/finite element spatial discretization approach with multigrid, fast multipole, tree code and/or preconditioned conjugate gradient solvers. For related work see Gholani et al. [40], from which it is clear that a careful choice of benchmarking solution is needed when comparing different discretization methods and elliptic system solvers.

A later study will describe how such model could be used to rank computing systems as a possible addition to the Linpack benchmark [52] and as an update of the NAS parallel benchmarks [26]. There are already several attempts to generate benchmarks to supplement the Linpack ranking of supercom-

puters, such as high performance conjugate gradient [24] and high performance geometric multigrid [5]. However, neither of these solve a real world problem or allow for algorithmic improvements in methods for solving differential equations and linear systems (see for example, Ballard et al. [28] and for Poisson's equation, Demmel [35, 36]). This paper aims to start a discussion on a simple method to rank supercomputers for solving realistic problems that also allows for algorithmic improvements, emphasizes the entire computer eco-system, including software that can be used and developed for that system, and the people that operate the system. Ramachandran et al. [57] have already noted that better ways to evaluate accelerators are required. By stating a more general problem than solution of a linear system of equations, one can use the best architecture specific algorithm on each platform thereby making a computer ranking not only a measurement of CPU double precision floating point power, but also of problem solving effectiveness.

1.1 The Klein-Gordon Equation

Nakanishi and Schlag [53] give an introduction to some of the theory of the Klein-Gordon equation, focusing primarily on the three-dimensional radial case. Two-dimensional simulations of the Klein-Gordon equation can be found in Bao and Yang [29] and Yang [65]. The linear Klein-Gordon equation occurs as a modification of the linear Schrödinger equation that is consistent with special relativity [47, 42]. At the present time, there have been no numerical studies of blow up of solutions to this equation without the assumption of radial symmetry. This equation has generated a large mathematical literature and yet is still poorly understood. Most of this mathematical literature has focused on analyzing the equation on an infinite three-dimensional space with initial data that either decays exponentially as one tends to infinity or is nonzero on a finite set of the domain. Here, we will simulate this equation in a periodic setting. Since this equation is a wave equation, it has a finite speed of propagation of information, much as a sound wave in air takes time to move from one point to another. Consequently for short time simulations, a simulation of a solution that is only nonzero on a finite part of the domain is similar to a simulation on an infinite domain. However, over long times, the solution can spread out and interact with itself on a periodic domain, whereas on an infinite domain, the interaction over long times is significantly reduced and the solution primarily spreads out. Understanding the interactions in a periodic setting is an interesting mathematical problem. Sufficiently smooth solutions of the Klein-Gordon equation conserve the energy given by

$$E(u, u_t) = \int \frac{1}{2} |u_t|^2 + \frac{1}{2} |u|^2 + \frac{1}{2} |\nabla u|^2 - \frac{1}{4} |u|^4 \, dx.$$

When accurate time stepping schemes are used for sufficiently bounded and differentiable solutions, the energy can act as a test of the correctness of the code implementation, the libraries used and of the computer hardware, since if the energy is not conserved and the implementation is correct, there is likely to be a hardware or library error.

2. NUMERICAL SCHEMES

The two time stepping schemes that were used by Donninger and Schlag [37] for radially symmetric solutions, can be readily adapted to fully three-dimensional simulations using Fourier pseudospectral discretization instead of a finite difference spatial discretization. One of these schemes is simple to implement. A modification of this scheme for implicit time stepping is described below because it is typical of numerical methods used for finding approximate solutions to partial differential equations and can be modified for use with other grid based spatial discretization methods.

2.1 A Second-Order Scheme

A modification of a second-order scheme used by Donninger and Schlag [37] and implemented in this study is

$$\frac{u^{n+1} - 2u^n + u^{n-1}}{\delta t^2} - \Delta \frac{u^{n+1} + 2u^n + u^{n-1}}{4} + \frac{u^{n+1} + 2u^n + u^{n-1}}{4} = |u^n|^2 u^n, \quad (2)$$

where $u^n \approx u(n\delta t, x, y, z)$. Time stepping takes place in Fourier space where the linear elliptic equation from the semi-implicit time discretization is easy to solve, and the three dimensional Fourier transform is used to obtain the non-linear term in real space. A more detailed explanation of the method and example programs can be found in Cloutier et al. [32], Rigge [58] and Balakrishnan et al. [27]. This scheme requires two FFTs per time step. The implementation used in this study allows for the field u to be complex.

3. RESULTS

We compare the scalability of the numerical scheme, without output to disk¹. In all cases the wall clock time for 30 time steps was measured. Figure 1 shows strong scaling results and Table 1 lists the computers according to the shortest run time, as well as documenting the properties of each computer.

3.1 Performance Model

Before discussing the results, it is helpful to have a model for how fast parallel computation can make this computation. Williams et al. [63] introduced the roofline model which provides a simple upper bound for performance of an algorithm at the node level. Fast Fourier transforms are typically bandwidth limited, the loop based computations in the code are performed on long vectors with unit stride accesses and little re-use of values loaded from memory, and so are similarly bandwidth limited at the node level. For each Fourier Transform, two `MPI_ALL_TO_ALL_V` or `MPI_ALL_TO_ALL` calls are used. For small message sizes, these are usually latency limited, while for large message sizes, these are usually network bandwidth limited. Since the problem size considered here is not too large, a simple model would use on chip bandwidth and network latency to determine an estimate of performance and scalability. Let a single core have a bandwidth of B_c , and the minimum

¹Except for the VSC2 for which no noticeable difference in runtime was observed when doing output to disk, and when not doing output to disk.

network latency for a single byte be L_n . For a discretization of size N^3 grid points, each time step requires approximately $d_1 \times N^3$ double precision floating point operations and $2d_2 \times [N \log(N)]^3$ operations for the FFT where d_1 and d_2 are constants². The constant d_1 is a processor dependent constant for doing vectorizable array floating point operations such as multiplying vectors by scalars and other vectors in the numerical solution of the partial differential equation, and d_2 is a processor and FFT library dependent constant for doing the one dimensional FFTs - in principle these can be given explicitly for each computer architecture and one dimensional FFT library. Hence, on a single core, the runtime is approximately given by $\frac{d_1 \times N^3 + 2d_2 \times [N \log(N)]^3}{B_c}$, so that on p processes the runtime is $\frac{d_1 \times N^3 + 2d_2 \times [N \log(N)]^3}{B_c \times p}$, assuming no network communication costs. When there is network latency, we get $\frac{d_1 \times N^3 + 2d_2 \times [N \log(N)]^3}{B_c \times p} + 2L_n$, for which there is a factor of 2 since two `ALL_TO_ALL` communications are needed. Thus, the network latency gives a lower bound on the possible speedup. It also takes time to send messages across a network. This depends on the topology of the network being used and the algorithm used to perform the `MPI_ALL_TO_ALL_V` or `MPI_ALL_TO_ALL` exchange [30, 41, 60, 62]. Since the problem size is not too large, it is reasonable to assume a small network dependent constant d_3 multiplied by the natural logarithm of the total number of processes (the lower bound for a hypercube network that is well suited to the FFT) so that we obtain $\frac{d_1 \times N^3 + d_2 \times [N \log(N)]^3}{B_c \times p} + 2L_n + d_3 \log(p)$. The model illustrates that for small p and fixed N , the runtime decreases close to linearly, and once p is large enough the runtime starts to increase again due to communication costs.

The model is similar though not as detailed as the ones in Ayala [25], Kerbyson and Barker [45] and Kerbyson et al. [46]. It is not possible to give a more detailed model within the scope of this paper for several reasons. First, modern FFT libraries such as ACML, FFTW and Intel MKL use auto tuning algorithms to pick appropriate FFT algorithms for a particular processor and FFT size. Second, 2DECOMP&FFT also uses auto tuning to optimize the domain decomposition that will give the shortest transpose time. The best domain decomposition depends on the network architecture which varies between supercomputers and may vary between jobs placed on different nodes of the same supercomputer. In particular, 2DECOMP&FFT will use the slab decomposition for a small number of process and the pencil decomposition for more than 512 processes, and since there are different algorithms for performing an all to all exchange, the best of which will depend on the size of the problem being solved, the computer being used and the number and location of the processors being used on that computer [39], it is not feasible to develop a more detailed model.

Since the computers used in this study are not hypercubes, the model can only serve as a proxy for a lower bound of the time to solution without communication and computation

²There is a factor of 2 because a forward and inverse transform is required by the algorithm

Rank	Machine Name	Time (s)	Cores used	Manufacturer and Model	Node Type	Total Cores	Interconnect	ID FFT Library	Chip Bandwidth Gb/s	Theoretical Peak TFLOP/s
1	Hornet [6]	0.319	12,288	Cray XC40	2x12 core Intel Xeon 2.5 GHz E5-2680v3	94,656	Cray Aries	FFTW 3 [20]	68	3,784
2	Juqueen [8]	0.350	262,144	IBM Blue Gene/Q	16 core 1.6 GHz Power PC A2	458,752	IBM 5D torus	ESSL [21]	42.6	5,872
3	Stampede [17]	0.581	8,162	Dell Power Edge	2x8 core Intel Xeon 2.7 GHz E5-2680	462,462	FDR infiniband	Intel MKL [7]	51.2	2,210
4	Shaheen [16]	1.66	16,384	IBM Blue Gene/P	4 core 0.85 GHz PowerPC 450	65,536	IBM 3D torus	ESSL [21]	13.6	222.8
5	MareNostrum III [9]	4.00	64	IBM DataPlex	2x8 core Intel Xeon 2.6 GHz E5-2670	48,384	FDR10 infiniband	Intel MKL [7]	51.2	1,017
6	Hector [4]	7.66	1024	Cray XE6	2x16 core AMD Opteron 2.3 GHz 6276 16C	90,112	Cray Gemini	ACML [19]	85	829.0
7	VSC2 [22]	9.03	1024	Megware	2x8 core AMD Opteron 2.2 GHz 6132HE	21,024	QDR infiniband	FFTW 3 [20]	42.8	185.0
8	Beacon [3]	9.13	256	Appro	2x8 core Intel Xeon 2.6 GHz E5-2670	768	FDR infiniband	Intel MKL [7]	51.2	16.0
9	Monte Rosa [10]	11.9	1,024	Cray XE6	2x16 core AMD Opteron 2.1 GHz 6272	47,872	Cray Gemini	ACML [19]	85	402.1
10	Titan [18]	17.0	256	Cray XK7	16 core AMD Opteron 2.2 GHz 6274	299,008	Cray Gemini	ACML [19]	85	2,631
11	Vedur [23]	18.6	1,024	HP ProLiant DL165 G7	2x16 core AMD Opteron 2.3 GHz 6276	2,560	QDR infiniband	FFTW 3 [20]	85	236
12	Aquila [2]	22.4	256	ClusterVision	2x4 core Intel Xeon 2.8 GHz E5462	800	DDR infiniband	FFTW 3 [20]	12.8	8.96
13	Neser [11]	118.7	128	IBM System X3550	2x4 core Intel Xeon 2.5 GHz E5420	1,024	Gigabit ethernet	FFTW 3 [20]	10.7	10.2

Table 1. A table showing ranking of speed at which different computers solve the Klein-Gordon equation for a grid size of 512^3 . The systems used, their compute chips, interconnect and underlying one dimensional FFT library used by 2DECOMP&FFT, theoretical chip bandwidth from RAM and theoretical peak floating point performance are also shown. Beacon, Stampede and Titan have accelerators which were not used in the study, hence their properties are not listed nor used in the calculation of theoretical peak double precision floating point performance.

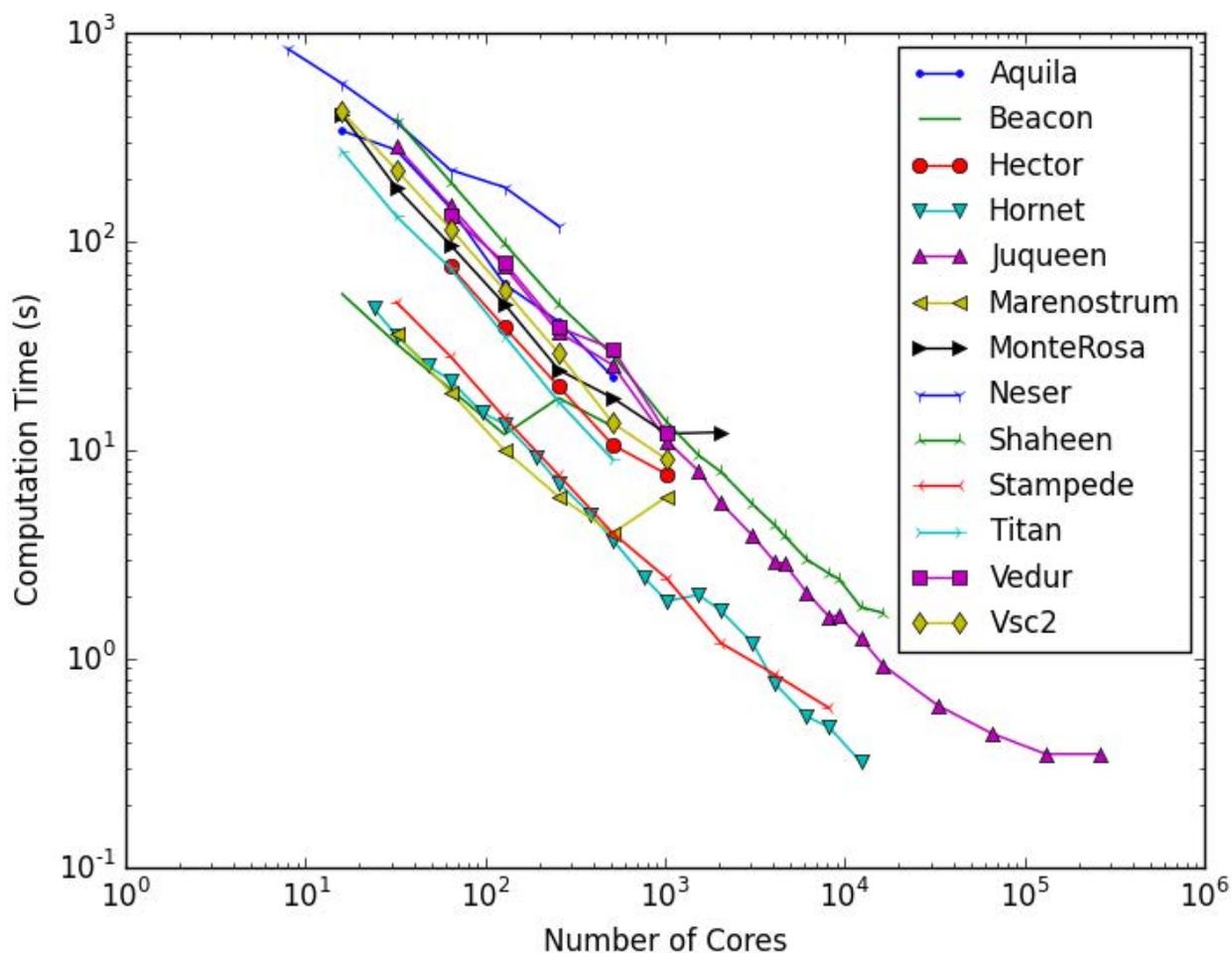


Figure 1. Scaling results showing computation time for 30 time steps as a function of the number of processor cores.

overlap. Obtaining a true lower bound with communication costs included is a significant undertaking. Given the large number of computers used, and the fact that even on a single computer, process placement, software environment configuration, as well as communications by other compute jobs would affect program performance, it is infeasible to give a more complete model for the communication cost.

It should be noted, that such a model may also be applicable to accelerated computers [31, 33, 34, 54, 59, 61]. The model does not account for overlap of computation and communication in computing a distributed Fourier transform. Distributed Fourier transforms which overlap computation and communication have been tried by Kandalla et al. [44] and Hoefler and Zerah [43] (MPI 3.0 non blocking collectives make this easier to implement), for which one might be able to find an improved lower bound.

The operation count used here is similar in spirit to that for solving the shallow water equations using a spectral transform method [64], but the Klein-Gordon equation is simpler,

and hence easier to use for evaluating computer performance. The equation also allows for a larger variety of solution methods to be used to solve linear system arising from discretizations of elliptic equations, and hence allows matching of an algorithm to the computer hardware.

3.2 Result Summary

Table 1 shows that for this problem, a ranking can be obtained that would allow for easy evaluation of parallel computers for solving partial differential equations of a given discretization in the fastest time possible using algorithms that are dominated by Fourier transforms. Figure 1 shows that the strong scaling limit is not reached on all the platforms primarily due to clusters being too small or due to queue size restrictions. For cases where less cores were used than close to full system size and the strong scaling limit was not reached, further computation time is required, in particular on Titan and Hornet. Results in the table can change with extensive tuning and careful job placement, however they are representative of what occurs in a typical production envi-

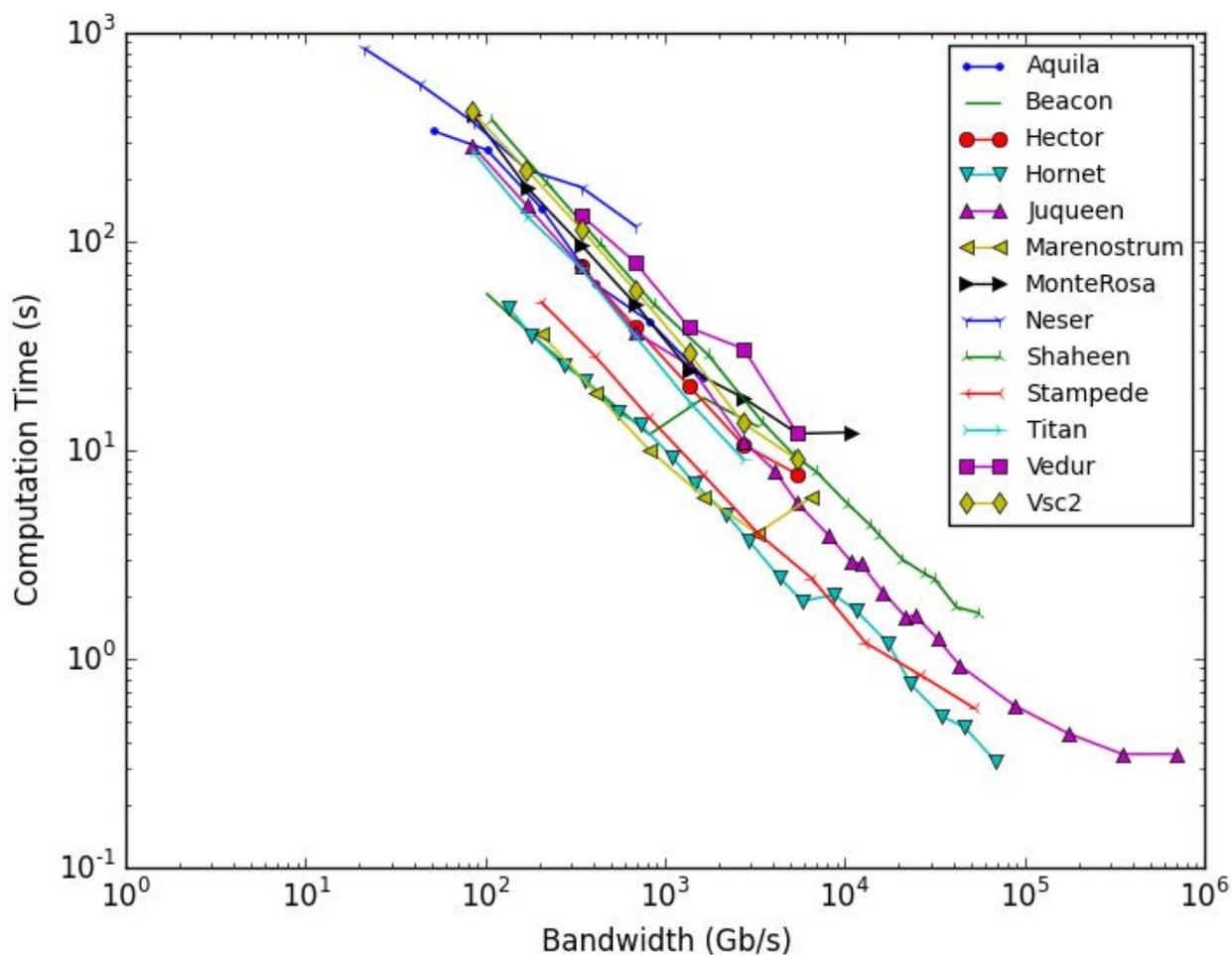


Figure 2. Scaling results showing computation time for 30 time steps as a function of total on chip bandwidth defined as the maximum theoretical bandwidth from RAM on a node multiplied by the number of nodes used.

ronment. Figure 2 shows scaling, but rather than using core count, uses total processor bandwidth which is defined as bandwidth per node multiplied by number of nodes used. Czechowski et al. [34] and Marjanovic et al. [51] indicate that node level bandwidth may be more important than system interconnect bandwidth, and indicate that for benchmarks which look at a fixed problem size, a few key performance indicators can replace the benchmark – for the current code, memory bandwidth and performance of `MPI_ALL_TO_ALL` are good indicators and for HPCG [24], Marjanovic et al. [51] indicate that memory bandwidth and `MPI_ALL_REDUCE` time are good performance indicators. Good performance for `MPI_ALL_TO_ALL` is much harder to achieve than good performance for `MPI_ALL_REDUCE` and implies good performance for `MPI_ALL_REDUCE`, though likely at a higher hardware cost.

3.3 Result Discussion

Table 1 shows that Hornet produces the fastest run time due to its high performance communication network and fast processors. Juqueen has a much higher theoretical peak floating point performance than Hornet. Juqueen’s large number of processor cores are difficult to use efficiently for this problem size, thus despite Hornet’s smaller size, it is more effective at solving the Klein Gordon equation using the current algorithm. Similar behavior is observed on Marenostrum III, where network performance makes it difficult to utilize a large portion of the machine despite the high theoretical peak performance.

On Aquila, there is a pronounced drop in performance when going from 8 to 16 cores, and after this the scaling is close to ideal again relative to the 16 core run (and the efficiency for 256 cores would be 76.5% if it was measured relative to the 16 core results). This drop in performance is likely because the 8 core run only requires intra-node communications, whereas all runs with higher core counts have to send

messages via the slower infiniband interconnect. A similar drop in performance is observed on Hornet in going from 2048 cores to 3072 cores, since MPI communication requires two communication steps on the dragon fly topology rather than just one communication step.

On Beacon, the runtime was quite sensitive to process placement, likely due to network topology. Nesar is the oldest machine used in this study. It is a Linux cluster with a 1 Gb ethernet. Nesar's low cost gigabit ethernet network prevents good scaling behavior to large core counts and explains its low ranking in table 1.

Speed up on Shaheen was very close to ideal due to the fast interconnect and balanced design which allows for good throughput to the cores given their maximum floating point performance. Shaheen has 0.85 GHz cores and Juqueen has cores that are clocked at 1.6 GHz and can do twice as many floating point operations per cycle, yet for the same number of cores, Juqueen is on average only 1.4 times faster than Shaheen, this is likely because each core on Juqueen has a lower share of bandwidth (2.66 Gb/s) than on Shaheen (3.375 Gb/s).

The performance on Vedur is significantly worse than on Hector, despite having similar compute chips. This is likely due to interconnect latency and slightly better integration on Cray computers. Beacon and Marenstrum share the same processors and have similar performance at small core counts, but network effects become important at larger core counts, so Marenstrum does better than Beacon overall.

Finally, figure 2 also shows that newer Intel processors on Beacon, Hornet, Marenstrum and Stampede give much better performance than the Power PC, older Intel and AMD processors for the same node level bandwidth. The reason for the improved performance is likely to be due to the larger number of floating point operations that can be done per cycle compared to the other architectures, and the more sophisticated memory controllers – this indicates that simply knowing chip bandwidth and performance of MPI_ALL_TO_ALL allows for a simple but incomplete model. For Juqueen, Marenstrum, MonteRosa and Nesar, the experiments with the largest number of cores do not scale anymore because communication costs start to dominate. As explained by Lo et al. [50], fitting a model to a machine is difficult to do precisely, so using models for predictions should be done with care.

4. SUMMARY

This study started with an extensive background to allow specialists in different areas (such as numerical solution of linear elliptic equations, mathematics, computer science) a context for understanding the paper and possible ways to either use the code or write their own codes and compare performance.

After introducing the Klein-Gordon equation and explaining why it is of interest, we present a simplified parallel runtime estimation model for solving the Klein-Gordon equation using parallelized FFTs on a distributed memory supercomputer. The proposed performance model aims to give an idea of scaling for a lower bound on time to solution. This could certainly be improved, but would require several papers to be done in greater generality and, for each machine it is possible

to find a lower bound for the best architecture specific algorithm. Space limitations prevent detailed models in this case. It should be noted that the current state of the art for parallel computational complexity counts do not realistically include architectures specific communication costs [28].

Timing results on 13 machines of various sizes are presented, from small machines with hundreds of cores to very large ones with hundreds of thousands of cores. The paper suggests these experiments are a better way to compare different parallel architectures than traditional benchmarks such as Linpack. A ranking of these computers has been proposed based on time to solve the Klein-Gordon equation. Different one dimensional FFT libraries are used to try and obtain optimal performance on each machine, and eventually it is hoped to use different linear solvers on each machine, since the aim is to obtain correct numerical results as efficiently (time or energy) as possible. One should therefore expect that the computer architecture will have an influence on the algorithm used.

The current benchmarking study has not examined speed of input and output. In parametric studies it is very helpful to visualize the solutions to understand how blow-up or dispersion occurs for which some input and output is needed. This visualization can either be done in situ or additionally (in particular when careful examination of the numerical solution is required) as a post-processing step. This can generate a large amount of input and output, which should also be a consideration for evaluating supercomputer performance.

5. FUTURE WORK

The results in this paper give a guide for codes which heavily utilize the Fourier transform on the different combinations of processor and interconnect that will give the best overall computation time. There are a variety of other methods for solving the same equation, and other aspects of using supercomputers that have not been covered in the present study, but would be useful to cover in other studies. Possible future work includes: in-situ visualization, finite difference/finite element codes using multigrid, tree code, fast multipole or conjugate gradient linear equation solvers for the implicit linear system solve in the time discretized Klein-Gordon equation, high order timestepping, measurement of computer energy consumption, use of accelerators, effectiveness of input and output. In addition more detailed performance models will be obtained on a smaller set of computers. Care will be required in choosing initial conditions to allow for a meaningful comparison with other ways of discretizing this equation and solving the linear equations in implicit time stepping schemes. It is our hope that this work, much like Donninger and Schlag [37] has done for us, will stimulate others to try their own implementations, compare them and improve upon this initial effort.

Acknowledgments.

The authors thankfully acknowledge the computer resources, technical expertise and assistance provided by: The Beacon Project at the University of Tennessee supported by the National Science Foundation under Grant Number 1137097; The UK's national high-performance computing service,

which is provided by UoE HPCx Ltd at the University of Edinburgh, Cray Inc and NAG Ltd, and funded by the Office of Science and Technology through EPSRC's High End Computing Programme; The Barcelona Supercomputing Center - Centro Nacional de Supercomputación; The Swiss National Supercomputing Centre (CSCS); The Texas Advanced Computing Center (TACC) at The University of Texas at Austin; The KAUST Supercomputer Laboratory (KSL) at King Abdullah University of Science and Technology (KAUST) for providing the resources that have contributed to this study; 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; The Aquila HPC service at the University of Bath; The Vienna Scientific Cluster (VSC); The PRACE research infrastructure resources in Germany at HLRS and FZ Jülich; The High Performance Computing Center of the University of Tartu. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding bodies or the service providers.

Initial research to start this project used resources of Kraken at the National Institute for Computational Science, Trestles at the San Diego Supercomputing Center both through the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1053575, the University of Michigan High Performance Computing Service FLUX and Mira at the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357

Oleg Batrašev and Benson Muite were partially supported by the Estonian Centre of Excellence in Computer Science (EXCS) under the auspices of the European Regional Development Funds. Mark Van Moer's contribution was made possible through the XSEDE Extended Collaborative Support Service (ECSS) program. We also thank the referees, Stefan Andersson, Winfried Auzinger, Andy Bauer, David E DeMarle, David Ketcheson, David Keyes, Robert Krasny, Dmitry Pekurovsky, Michael Pippig, Paul Rigge, Madhusudhanan Srinivasan, Eero Vainikko, Matthias Winkel, Brian Wylie, Hong Yi and Rio Yokota for helpful advice and suggestions.

6. REFERENCES

1. 2decomp&fft. <http://2decomp.org/>, November 2014.
2. Aquila cluster. <https://wiki.bath.ac.uk/display/HPC/Aquila>, November 2014.
3. Beacon cluster. <https://www.nics.tennessee.edu/beacon>, November 2014.
4. Hector supercomputer. <http://www.hector.ac.uk/>, November 2014.
5. High performance geometric multigrid. <https://hpgmg.org/>, November 2014.
6. Hornet supercomputer. <http://www.hlrs.de/systems/platforms/cray-xc40-hornet/>, November 2014.
7. Intel math kernel library. <https://software.intel.com/en-us/intel-mkl/>, November 2014.
8. Juqueen supercomputer. <http://www.fz-juelich.de/ias/jsc/juqueen>, November 2014.
9. Marenostrum supercomputer. <http://www.bsc.es/marenostrum-support-services>, November 2014.
10. Monte rosa supercomputer. http://www.cscs.ch/computers/monte_rosa, November 2014.
11. Nesar cluster. http://www.hpc.kaust.edu.sa/documentation/user_guide/resources/nesar/, November 2014.
12. Openfft. <http://www.openmx-square.org/openfft/>, November 2014.
13. P3dfft. <https://code.google.com/p/p3dfft/>, November 2014.
14. Pfft. <https://www-user.tu-chemnitz.de/~mpip/software.php#pfft>, November 2014.
15. Pkufft. https://code.google.com/p/parray-programming/wiki/Chapter_7_Case_Studies, November 2014.
16. Shaheen. <http://ksl.kaust.edu.sa/Pages/Shahen.aspx>, November 2014.
17. Stampede supercomputer. <https://www.tacc.utexas.edu/user-services/user-guides/stampede-user-guide>, November 2014.
18. Titan supercomputer. <https://www.olcf.ornl.gov/computing-resources/titan-cray-xk7/>, November 2014.
19. AMD Core Math Library. <http://developer.amd.com/tools-and-sdks/cpu-development/amd-core-math-library-acml/>, November 2014.
20. Fast Fourier Transform in the West. <http://fftw.org/>, November 2014.
21. IBM Engineering and Scientific Subroutine Library. <http://www-03.ibm.com/systems/power/software/essl/>, November 2014.
22. VSC2 cluster. <http://vsc.ac.at/about-vsc/vsc-pool/vsc-2/>, November 2014.
23. Vedur cluster. http://www.hpc.ut.ee/en/vedur_cluster, November 2014.

24. High performance conjugate gradient. <http://hpcg-benchmark.org/>, January 2015.
25. Ayala, O., and Wang, L.-P. Parallel implementation and scalability analysis of 3D fast Fourier transform using 2D domain decomposition. *Parallel Computing* 39 (2013), 58–77.
26. Bailey, D. *Encyclopedia of Parallel Computing*. Springer, 2011, ch. NAS Parallel Benchmarks., 1254–1259.
27. Balakrishnan, S., Bargash, A., Chen, G., Cloutier, B., Li, N., Muite, B., Quell, M., Rigge, P., Solimani, M., Souza, A., Thiban, A., West, J., Malicke, D., Van Moer, M., and San Roman Alerigi, D. Parallel spectral numerical methods. http://en.wikibooks.org/w/index.php?title=Parallel_Spectral_Numerical_Methods, November 2014.
28. Ballard, G., Carson, E., Demmel, J., Hoemmen, M., Knight, N., and Schwartz, O. Communication lower bounds and optimal algorithms for numerical linear algebra. *Acta Numerica* (2014), 1–155.
29. Bao, W., and Yang, L. Efficient and accurate numerical methods for the Klein-Gordon-Schrodinger equations. *J. Comp. Phys.* 225, 2 (2007), 1863–1893.
30. Bertsekas, D., Özveren, C., Stamoulis, G., Tseng, P., and Tsitsiklis, J. Optimal communication algorithms for hypercubes. *Journal of Parallel and Distributed Computing* 11 (1991), 263–275.
31. Chen, Y., Cui, X., and Mei, H. Large-scale FFT on GPU clusters. In *Proc. International Conference on Supercomputing* (2010).
32. Cloutier, B., Muite, B., and Rigge, P. Performance of FORTRAN and C GPU extensions for a benchmark suite of Fourier pseudospectral algorithms. In *Proc. Symposium on Application Accelerators in High Performance Computing (SAAHPC)* (2012), 1145–1148.
33. Czechowski, K., McClanahan, C., Battaglini, C., Iyer, K., Yeung, P.-K., and Vuduc, R. Prospects for scalable 3D FFTs on heterogeneous exascale systems. In *Proc. ACM/IEEE Conf. Supercomputing (SC)* (2011).
34. Czechowski, K., McClanahan, C., Battaglini, C., Iyer, K., Yeung, P.-K., and Vuduc., R. On the communication complexity of 3D FFTs and its implications for exascale. In *Proc. ACM Int'l. Conf. Supercomputing (ICS)* (2012).
35. Demmel, J. *Applied Numerical Linear Algebra*. SIAM, 1997.
36. Demmel, J. Applications of parallel computing. www.cs.berkeley.edu/~demmel/cs267_Spr14/Lectures/lecture21_structured_jwd14_4pp.pdf, 2014.
37. Donninger, R., and Schlag, W. Numerical study of the blowup/global existence dichotomy for the focusing cubic nonlinear Klein-Gordon equation. *Nonlinearity* 24 (2011), 2547–2562.
38. Duy, T., and Ozaki, T. A decomposition method with minimum communication amount for parallelization of multi-dimensional FFTs. *Computer Physics Communications* 185 (2014), 153–164.
39. Foster, I., and Worley, P. Parallel algorithms for the spectral transform method. *SIAM J. Sci. Stat. Comput.* 18, 3 (1997), 806–837.
40. Gholami, A., Malhotra, D., Sundar, H., and Biros, G. FFT, FFM or multi grid? A comparative study of state-of-the-art Poisson solvers. arXiv:1408.6497, 2014.
41. Grama, A., Gupta, A., Karypis, G., and Kumar, V. *Introduction to Parallel Computing.*, 2nd ed. Addison Wesley, 2003.
42. Grennner, W. *Relativistic Quantum Mechanics*. Springer, 1984.
43. Hoefler, T., and Zerah, G. Optimization of a parallel 3D-FFT with non-blocking collective operations. 3rd International ABINIT Developer Workshop, 2007.
44. Kandalla, K., Subramoni, H., Tomko, K., Pekurovsky, D., Sur, S., and Panda, D. High-performance and scalable non-blocking all-to-all with collective offload on infiniband clusters: a study with parallel 3D FFT. *Comput. Sci Res Dev* 26 (2011), 237–246.
45. Kerbyson, D., and Barker, K. A performance model of direct numerical simulation for analyzing large-scale systems. In *Proc. Workshop on Large Scale Parallel Processing, IEEE International Parallel and Distributed Processing* (2011).
46. Kerbyson, D., Barker, K., Gallo, D., Chen, D., Brunheroto, J., Ryu, K., Chiu, G., and Hoisie, A. Tracking the performance evolution of Blue Gene Systems. In *Proc. ISC 13, Lecture Notes in Computer Science*, Springer Verlag (2013), 317–329.
47. Landau, R. *Quantum Mechanics II*. Wiley, 1996.
48. Li, N., and Laizet, S. 2DECOMP&FFT - A highly scalable 2D decomposition library and FFT interface. In *Proc. Cray User Group* (2010).
49. Li, Q., Ji, Z., Zheng, Z., and Liu, H. Numerical solution of nonlinear Klein-Gordon equation using Lattice Boltzmann method. *Applied Mathematics* 2 (2011), 1479–1485.
50. Lo, Y., Williams, S., Straalen, B., Ligocki, T., Cordery, M., Oliker, L., Hall, M., and Wright, N. Roofline model toolkit: A practical tool for architectural and program analysis. In *Proc. 5th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS14)* (2014).
51. Marjanovic, V., Garcia, J., and Glass, C. Performance modeling of the HPCG benchmark. In *Proc. 5th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS14)* (2014).

52. Meuer, H., Strohmaier, E., Dongarra, J., Simon, H., and Meuer, M. Top 500 list. <http://www.top500.org/>, November 2014.
53. Nakanishi, K., and Schlag, W. *Invariant Manifolds and Dispersive Hamiltonian Evolution Equations*. European Mathematical Society, 2011.
54. Park, J., Bikshandi, G., Vaidyanathan, K., Tang, P., Dubey, P., and Kim, D. Tera-scale 1D FFT with low-communication algorithm and Intel Xeon®Phi™ coprocessors. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*, no. 34 (2013).
55. Pekurovsky, D. P3DFFT: A framework for parallel computations of Fourier transforms in three dimensions. *SIAM J. Sci. Comput.* 34 (2012), C192–C209.
56. Pippig, M. PFFT: An extension of FFTW to massively parallel architectures. *SIAM J. Sci. Comput.* 35, 3 (2013), C213–C236.
57. Ramachandran, A., Vienne, J., Vand Der Wijngaart, R., Koesterke, L., and Sharapov, I. Performance evaluation of NAS parallel benchmarks on Intel Xeon Phi. In *Proc. 42nd International Conference on Parallel Processing* (2013).
58. Rigge, P. Numerical solutions to the Sine-Gordon equation. arXiv:1212.2716, 2012.
59. Song, S., and Hollingsworth, J. Designing and auto-tuning parallel 3-D FFT for computation-communication overlap. In *Proc. of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming* (2014), 181–192.
60. Swarztrauber, P. Multiprocessor FFTs. *Parallel Comput.* 5 (1987), 197–210.
61. Swarztrauber, P. Multipipeline multiprocessor system. <http://www.google.com/patents/US5689722>, 1997.
62. Swarztrauber, P., and Hammond, S. A comparison of optimal FFTs on torus and hypercube multicomputers. *Parallel Computing* 27 (2001), 847–859.
63. Williams, S., Waterman, A., and Patterson, D. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM* 53, 4 (2009), 65–76.
64. Worley, P., and Foster, I. Parallel spectral transform shallow water model: a runtime-tunable parallel benchmark code. In *Proc. of the Scalable High Performance Computing Conference*, J. Dongarra and D. Walker, Eds. (1994), 207–214.
65. Yang, L. Numerical studies of the Klein-Gordon-Schrödinger equations. Master's thesis, National University of Singapore, 2006.

Biography

Samar Aseeri is a computational scientist at the King Abdullah University of Science and Technology.

Oleg Batrašev is a researcher in computer science at Tartu Ülikool.

Matteo Icardi is a postdoctoral research fellow at the King Abdullah University of Science and Technology.

Brian Leu is an undergraduate student in electrical engineering at the University of Michigan.

Albert Liu is an undergraduate student in electrical engineering and physics at the University of Michigan. Visualizations may be found at <http://www.umich.edu/~alberliu/kleingordon.html>.

Ning Li is a high performance computing consultant at the Numerical Algorithms Group. Ning specializes in numerical software development, in particular in the area of high performance computing and wrote the parallel software framework 2DECOMP&FFT.

Benson Muite is a postdoctoral research fellow in computer science at Tartu Ülikool.

Eike Müller is a postdoctoral research associate in mathematics at the University of Bath.

Brock Palen is a high performance computing system administrator at the University of Michigan.

Michael Quell is an undergraduate student in mathematics at the Technische Universität Wien. Visualizations may be found at <http://youtu.be/rNRSBTiLYHE>.

Harald Servat is a computer science researcher at the Barcelona Supercomputing Center.

Parth Sheth is a research assistant in biomedical engineering at the University of Michigan.

Robert Speck is a mathematician at Forschungszentrum Jülich, Jülich Supercomputing Centre.

Mark Van Moer is a visualization scientist at the National Center for Supercomputing Applications.

Jerome Vienne is a computational scientist at the Texas Advanced Computational Center.

A Self-Adaptive Method for Frequent Pattern Mining using a CPU-GPU Hybrid Model

Lan Vu, Gita Alaghband
University of Colorado Denver
{lan.vu, gita.alaghband}@ucdenver.edu

ABSTRACT

Frequent pattern mining (FPM) is an important and computationally intensive task in data mining. We present a novel method, CGMM (CPU & GPU based Multi-strategy Mining), for mining frequent patterns that combines the computing power of CPU and GPU to speed up the frequent pattern mining. CGMM employs two different mining strategies and dynamically switches between them; the CPU-based strategy uses FP-tree data structure to perform the mining task on CPU while the GPU-based method converts the allocated data portions to bit vectors to work mainly on GPU. This unique approach has the following advantages compared to the existing methods: (1) utilizes the parallel processing capability of GPU for computationally intensive portions; the flexibility and low memory latency of CPU for the sophisticated data processing needed to manipulate the more complex data structures to enhance the overall performance (2) applies two mining strategies to efficiently mine both sparse and dense databases. The performance evaluation of CGMM on a machine with AMD CPUs and NVIDIA Tesla GPUs shows that in the best cases, the proposed method runs up to 229 times faster than well-known sequential FPM algorithms and 7.2 - 13.9 times faster than GPa priori, a GPU based algorithm for FPM. In addition to outperforming them, CGMM has more stable performance on both dense and sparse test datasets.

Author Keywords

Guides; instructions; author's kit; conference publication; keywords should be separated by a semi-colon.

INTRODUCTION

General-Purpose Graphics Processing Units (GPGPU) have emerged as powerful computing resources for general-purpose computing applications. They are used as co-processors capable of fast intensive computational processing that was once performed by CPUs [1, 2].

HPC 2015, April 12 - 15, 2015, Alexandria, VA, USA

© 2015 Society for Modeling & Simulation International (SCS)

GPGPU applications are implemented using either CUDA [3] or OpenCL [4]. As the volume of data generated in most fields is fast growing, applying high performance techniques to enhance the overall performance of data analysis tasks has become important. Frequent pattern mining (FPM) is a crucial component of data mining used to find various types of relationships among variables in large databases such as associations [5], correlations [6], causality [7], sequential patterns [8], episodes [9] and partial periodicity [10] and has many practical applications such as market analysis, biomedical and computational biology, web mining, decision support, telecommunications alarm diagnosis and prediction, and network intrusion detection [11, 12].

Motivation

Most existing FPM methods for GPU [13 - 22] are derived from Apriori [5], a sequential method that uses breadth-first strategy and the candidate generation-and-test approach to find frequent patterns as well as utilizes *downward closure* property (i.e. a k-itemset is frequent only if all of its sub-itemsets are frequent) to sharply reduce the search space. These features allow the GPU-based methods which create a large amount of workload with data suitable for presentation on GPU. However, for very large databases, the Apriori-like methods are significantly slower and consume much more memory in comparison to sequential methods like Eclat [23], FP-growth [24], and especially our proposed methods, FEM [25] and DFEM [26]. In many cases, Apriori is hundred times slower than FP-growth or its variants the FP-tree traversal time [27], H-mine [28], nonordfp [29], the use of FP-array data structure [30] and FP-growth with database partition projection [31]. Huang et al. [19] have shown that a parallel Apriori algorithm on GPU even run slower than sequential FP-growth run on CPU. Therefore, better mining strategy is essential. Additionally, Apriori performs efficiently on sparse data but not on dense databases [32, 33, 34], the GPU based Apriori methods for FPM exhibit the same poor performance on dense datasets as well.

Contribution

We propose CGMM for FPM which utilizes both CPU and GPU for its computation. The following features of CGMM contribute to its improved performance and distinguish it from the prior FPM methods:

- 1) CGMM consists of two different mining strategies, *CPUBasedMining* and *GPUBasedMining*, specifically designed to exploit the computing power of both CPU

and GPU. It uses a heuristic approach to dynamically select a suitable strategy for each data subset of the database based on its density characteristics during the execution.

- 2) The *CPUBasedMining* strategy uses only CPU to mine the frequent patterns by recursively constructing FP-trees without generating a large number of candidates. It is applied for data portions with sparse characteristics.
- 3) The *GPUBasedMining* strategy uses GPU as the main computing engine to mine the data portions with dense characteristics using a hybrid solution that consists of a new adaptive breadth-first approach, bit vector data structures, and candidate generation and test approach.

BACKGROUND

Problem Statement

The FPM problem is defined as follows: Let $I = \{i_1, i_2, \dots, i_n\}$ be the set of all distinct items in the transactional database D . The *support* of an *itemset* α , a set of items, is the percentage of transactions containing α in D . A k -*itemset* α , which consists of k items from I , is frequent if α 's *support* is larger or equal to *minsup*, where *minsup* is a user-specified minimum support threshold. Given a database D and a *minsup*, FPM searches for the complete set of frequent itemsets in D . For example, given the database in Table 1 and *minsup*=20%, the frequent 1-itemsets include a, b, c, d and e while f is infrequent because the *support* of f is only 11%. Similarly, $ab, ac, ad, ae, bc, bd, cd, ce, de$ are frequent 2-itemsets and abc, abd, ace, ade are the frequent 3-itemsets.

Table 1. Sample dataset with *minsup* = 20%

Transaction ID	Items	Sorted Frequent Items
1	b,d,a	a,b,d
2	c,b,d	b,c,d
3	c,d,a,e	a,c,d,e
4	d,a,e	a,d,e
5	c,b,a	a,b,c
6	c,b,a	a,b,c
7	f	
8	b,d,a	a,b,d
9	c,b,a,e,f	a,b,c,e

Mining Frequent Pattern Using GPU

Mining frequent patterns is nontrivial because of its exponential search space, large amount of data and computational intensity. In the trend of applying high performance computing to increase the processing speed of data analysis tasks, developing FPM methods for GPU has received much interest because of the massive parallel capability of this device.

Modern GPUs have between dozens to hundreds of computing units/cores used as co-processors and can deliver much larger performance than a CPU for the right type of application. Their architectures typically consist of several streaming multiprocessors sharing same device memory. Each multiprocessor can have eight or more computing

units/cores depending on the device model. The following general steps are needed for an application to run on a GPU: (1) copy input data from the main memory of CPU to the device (GPU) memory; (2) perform the computation on GPU; (3) copy the output data back from the device memory to the main memory. Two popular software platforms used to develop applications for GPU are CUDA [3] and OpenCL [4]. In CUDA computation is written as a *kernel* on the CPU to be launched on GPU with a massive amount of similar computational units known as *threads* [13]. A *kernel* launched to execute on GPU is referred to as a *grid*. A *grid* consists of multiple *thread blocks* where each block is a group of *threads*. A *thread block* is assigned to a multiprocessor whose execution is in the form of SIMT (Single Instruction, Multiple Threads). The computing model of OpenCL is similar to CUDA. The GPU SIMT model of computation (which for most practical purposes is very similar to SIMD: Single Instruction, Multiple Data) works great for applications with massive amount of repetitive parallelism with regular access patterns. Performance penalties occur on GPUs due to irregular workloads, irregular access patterns, need for synchronization among threads belonging to different blocks, and need to move and access data in different levels of memory hierarchy.

The traditional FPM methods developed for sequential execution on CPU must be redesigned so that their computational model and data structure can adapt well to the GPU architecture. This is a highly challenging task for many of the complex applications designed for general purpose computing including FPM because GPU requires data presentation that can be processed uniformly and independently by a large number of concurrent *threads*. In addition, data pre/post processing in CPU and transferring data between CPU and GPU can add a large enough overhead to the total execution time of FPM to negate the benefits of GPU.

Prior GPU-based FPM Algorithms

In this section, we review three most relevant GPU FPM algorithms CSFPM [17], GPAPriori [20] and gpuDCI [18]

CSFPM (Candidate Slicing Frequent Pattern Mining) [17] is an Apriori-like method for GPU. It off-loads the most time consuming phase of counting to compute the supports to the GPU to speed up the total execution time. For better load balancing, the algorithm parallelizes and distributes the candidate itemsets to the GPU threads; each thread checks its own transaction in a candidate item. This reduces the processor waiting time since the load between processing units is more balanced.

GPAPriori [20] is also an Apriori-like method for GPU. It maps the Apriori algorithm to the SIMD execution model by using bitset to represent the input database where the i^{th} bit in a bitset presents the occurrence of that item/itemset in the i^{th} transaction of database (1: exist, 0: does not exist). This data structure improves upon the traditional approach of the vertical data layout in state-of-the-art Apriori implementations. Similar to CSFPM, GPAPriori parallelizes only support counting step on the GPU while the remaining

steps are executed on CPU. GPApriori applies several optimization techniques in its implementation: (1) before support counting is performed on GPU, candidates are preloaded to shared memory to prevent repeating global memory reads, (2) manual, hand-tuned loop unrolling to further improve the kernel speed; and (3) hand-tuned block size [20].

gpuDCI [18] is adapted from the DCI algorithm [35], a sequential mining approach that combines Apriori and Eclat. This algorithm starts its computation on CPU, as in DCI, and moves the pruned datasets to the GPU as soon as the bitwise vertical dataset fits into the GPU global memory. Afterwards the support computation is performed on GPU. However, after switching to GPU, the CPU still manages patterns, generates candidates and stores patterns that are frequent according to the supports computed by the GPU. Two parallel techniques were investigated: (1) for the transaction-wise technique, all GPU cores independent of the GPU multiprocessor they belong to, work on the same intersection or count operation; (2) for the candidate-wise technique, each GPU multiprocessor intersects and counts a different candidate. The candidate-wise technique has shown to perform better than the transaction-wise technique because it requires fewer synchronization operations.

NEW FREQUENT PATTERN MINING APPROACH USING A CPU-GPU HYBRID MODEL

Among many sequential frequent pattern mining methods that are traditionally developed for machines without GPU, FP-growth and its variants [27 – 31] are most efficient, especially for sparse large databases. They do not require generating a very large number of candidate itemsets as Apriori and Eclat do and hence save both memory and computation. The main mining computation of FP-growth is based on recursively generating FP-trees [24].

While the benefits of applying FP-growth cannot be ignored, developing a method based on FP-growth for FPM poses a lot of challenges for GPU due to FP-tree data structure, recursive tree construction, and tree traversal need. GPUs perform best for tasks whose data structures are linear and computations lend themselves well to vector processing. Moreover, it has been shown that FP-growth does not perform as well as Eclat when mining dense databases or mining with low minsups where the number of generated frequent pattern is very large [33, 34, 36]. For such cases, manipulating a very large number of FP-trees in FP-growth becomes more costly than intersecting the TID-lists of Eclat - the vertical layout of database where each list of a item/itemset stores IDs of transactions containing that item/itemset. It is important to keep in mind that for TID-lists, the vertical and linear data formats and list intersection operations of Eclat are quite suitable for GPU but the depth-first approach does not allow creation of enough parallel workload, compared to Apriori, to fully utilize the large computing resources of GPU.

Therefore, we combine and redesign the advanced features of FP-growth, Eclat, MAFIA [12] and Apriori into a new mining method, named CGMM, that applies both CPU and GPU computing to provide high FPM performance. As in DFEM, CGMM consists of two mining strategies and dynamically selects a suitable mining for each data portion of a database.

Overview of CGMM

CGMM consists of three main tasks: FP-tree construction, *CPUBasedMining* and *GPUBasedMining* described below. It starts with constructing the corresponding FP-tree followed by dynamically selecting between *CPUBasedMining* and *GPUBasedMining* similar to DFEM as depicted in Figure 1.

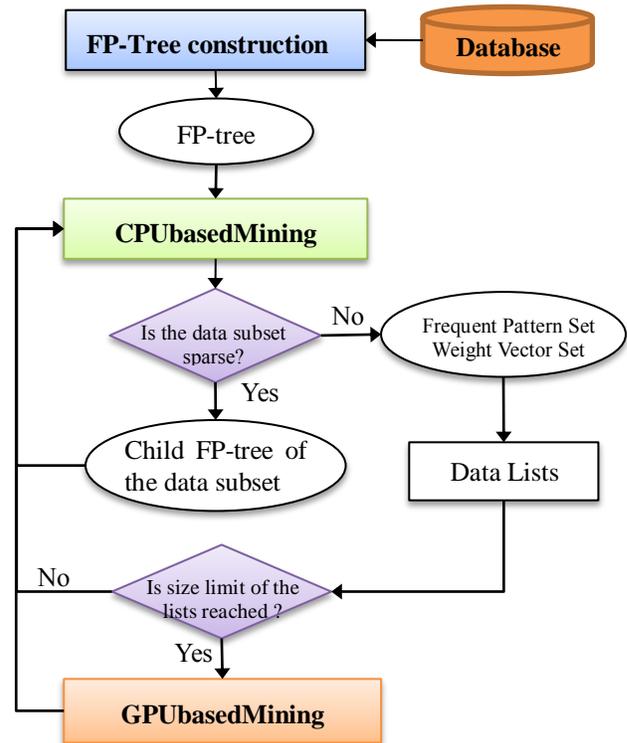


Figure 1: The overview of CGMM

FP-tree construction (on CPU) reads the database to build the corresponding FP-tree. Use of this data structure significantly reduces the I/O cost because it compacts the database in memory before mining is performed. It also enables multiple mining strategies to be employed with relative ease as subsets of data from this tree can be used to create independent sub-mining tasks where each may use a different mining strategy.

CPUBasedMining (on CPU) extracts from the FP-tree the data subsets to recursively construct child FP-trees. The frequent patterns are identified based on newly generated FP-trees without the requirement of generating a large number of frequent pattern candidates. Because FP-tree data structure is complex and inefficient for mining on

GPU, only CPU is used to process the mining task. *CPUBasedMining* is distinct from the other mining methods because it is applied to sparse data subsets only. Determination of whether a data subset is sparse or dense is described in following section.

GPUBasedMining uses a new hybrid mining model designed for GPU applied to the dense data subsets. It presents data used to compute the *support* as bit vectors and maintains input and output data in data lists including *Frequent Pattern Set List* and *Weight Vector List*. The new frequent patterns are generated by applying candidate generation-and-test approach using a self-adaptive breath-first solution. Only a subset of frequent patterns is used to generate frequent pattern candidates at a time as long as their input and output data fit in the GPU memory. It addresses GPU memory limitation problem. Unlike the existing GPU solutions that off-load only the *support* counting phase to GPU, *GPUBasedMining* performs both the candidate generation and the *support* counting on GPU to increase GPU utilization and reduce overall processing on CPU as well as data transfer between CPU and GPU memories.

CGMM ALGORITHM

The CGMM algorithm (Figure 2) is performed in two stages. The first stage is loading data into memory by constructing the FP-tree. Then, frequent patterns are generated using the two mining strategies in our algorithms by initially invoking *CPUBasedMining*.

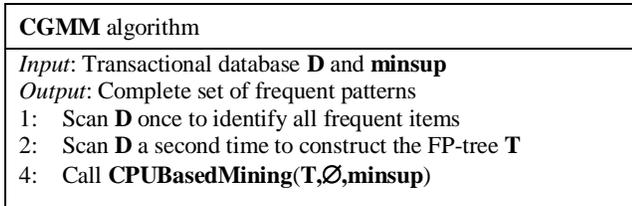


Figure 2: CGMM algorithm

FP-tree Construction

FP-tree is a prefix tree that compacts all sets of ordered frequent items from database into memory. This tree consists of a header table storing the frequent items with their *count*, a root node and a set of prefix sub-trees. Each node of the tree includes an *item name*, a *count* indicating the number of transactions that contain all items in the path from the root node to the current node, and a *link* to its parent node. Each linked list starting from the header table links all nodes of the same frequent item. If two itemsets share a common prefix, the shared part can be merged as long as the *count* properly reflects the frequency of each itemset in the database.

The construction of FP-tree requires two database scans. Only CPU is used for this stage because the tree data structure and operations are not suitable for GPU computing. Database is scanned the first time to find the frequent items and create the header table. A second

database scan is done to get frequent items of each transaction. Next, these items are sorted and inserted in the FP-tree in frequency descending order. During the top-down traversal of the tree construction, if a node presenting an item exists, its count will be incremented by one. Otherwise, a new node is added to the FP-tree. Figure 3 illustrates an FP-tree constructed from the dataset in Table 1 where a pair $\langle x:y \rangle$ indicates *item name* and its *count*.

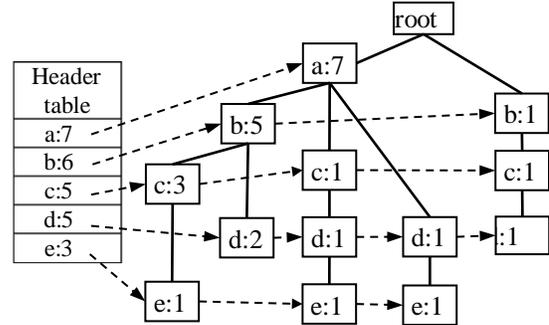


Figure 3: FP-tree constructed from the database in Table 1

CPUBasedMining

CPUBasedMining initializes the process of generating frequent patterns and mines the sparse data portions of the database. Similar to FP-tree construction stage, only CPU is used in this stage. The frequent patterns are reported by concatenating the suffix pattern of the previous step with each item α of the input FP-tree. At the beginning, this suffix pattern is \emptyset . Then, CGMM constructs a child FP-tree called conditional FP-tree for every item α using a data subset called conditional pattern base. This data subset is extracted from the input FP-tree of each recursive iteration and consists of sets of frequent items co-occurring with the suffix pattern. For example, the conditional pattern base of item d, which is extracted from the FP-tree (Figure 3) by bottom-up traversal starting from the nodes in the linked list of item d, consists of the 4 sets $\{a:2, b:2\}$, $\{a:1, c:1\}$, $\{a:1\}$ and $\{b:1, c:1\}$ in which $\{a, b\}$ occurs twice (Figure 4-a). This base is used to construct the conditional FP-tree (Figure 4-b). The new tree is then used as the input of the next step of recursive iteration of this mining task.

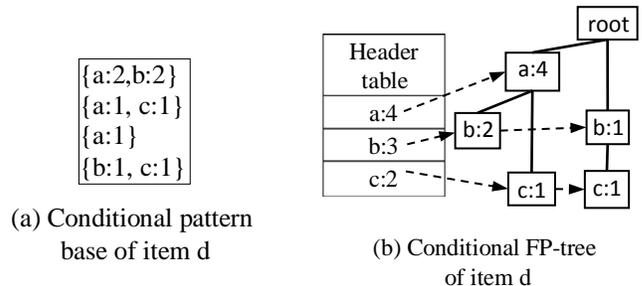


Figure 4: FP-tree constructed from the conditional pattern base of item d

CPUBasedMining does not process data subsets which have dense characteristics. Instead, it converts them into *Frequent Pattern Set* and *Weight Vector* and adds them into the *Frequent Pattern Set List* and *Weight Vector List* managed by GPUBasedMining. As the mining proceeds and when CPUBasedMining finds these lists full, it invokes GPUBasedMining to start the mining process on GPU. The algorithmic description of CPUBasedMining is in Figure 5.

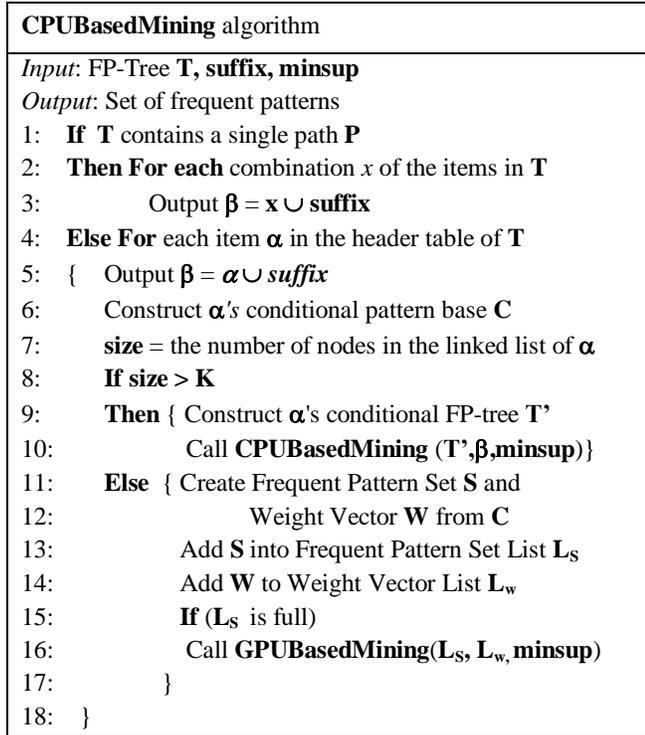


Figure 5: CPUBasedMining algorithm

GPUBasedMining

GPUBasedMining mines the dense data portions of the database. It uses the GPU as co-processor for its most computational intensive need and CPU for the complicated tasks with data dependence to exploit the power and flexibility of GPU and CPU respectively.

Data Structures

GPUBasedMining uses several data structures to manage the mining data including *Frequent Pattern Set*, *Frequent Pattern Set List* and *Weight Vector List* (Figure 6).

Frequent Pattern Set is a set of frequent patterns that have same length k (i.e. they have k items in their itemsets) in which $(k-1)$ items are common and one item is different among the frequent patterns in the set. For example, three frequent patterns abc, abd, abe can form a *Frequent Pattern Set* because they have ab in common. It contains a set of bit vectors where each presents the occurrence of a frequent pattern in the database. This data structure is used to generate new $(k+1)$ length frequent pattern candidates and compute their *supports*. Some additional information of a *Frequent Pattern Set* includes *size* - the number of frequent

patterns in the set, *length* - the number of items in a frequent pattern. Figure 6 demonstrates two *Frequent Pattern Sets* that are created from the conditional pattern bases of items e and c extracted from the FP-tree (Figure 3). A *Frequent Pattern Set* is only created if it satisfies the condition for GPUBasedMining and contains the data of at least two frequent patterns.

Frequent Pattern Set List works as a container that holds all *Frequent Pattern Sets* generated during the mining process and is updated by both CPUBasedMining and GPUBasedMining. In our example (Figure 6), two *Frequent Pattern Sets* are added into the *Frequent Pattern Set List*.

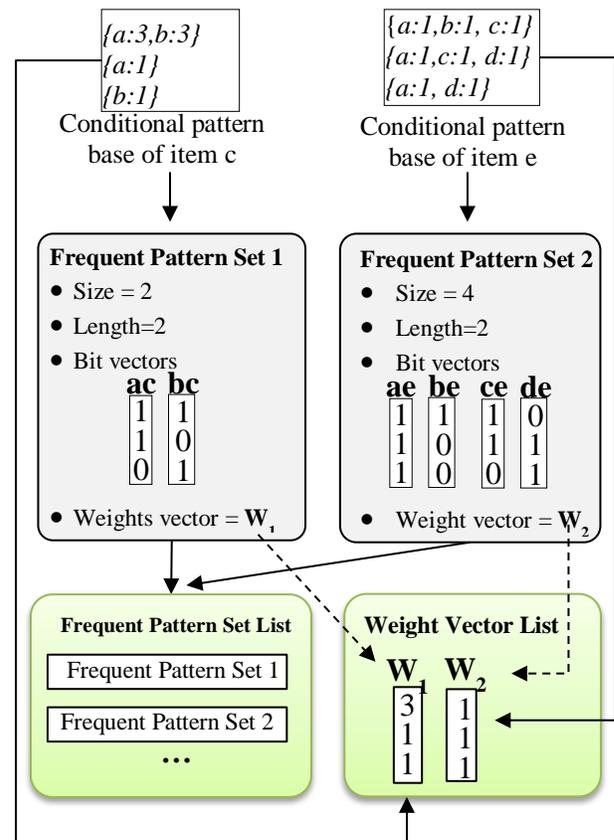


Figure 6: Data structures used by GPUBasedMining

Weight Vector List: the weight vector is a portion of *Frequent Pattern Set* that is used to compute the *support* of the patterns and is created by collecting the frequency values of sets in the conditional pattern base. Because many *Frequent Pattern Sets* that are newly generated by GPUBasedMining may share a same weight vector, we store this data in a separate list called *Weight Vector List* and add a reference in *Frequent Pattern Set* to its *weight vector* in the list to avoid duplication, save memory.

Algorithmic Descriptions

During the execution of CPUBasedMining, small data subsets that meet the condition to be mined using GPUBasedMining are converted to *Frequent Pattern Sets*

and added into the *Frequent Pattern Set List*. When the number of items in this list reaches a predefined limit, GPUBasedMining is invoked by CPUBasedMining to start generating all new frequent patterns using *Frequent Pattern Sets* in the list. We present experiments showing the impact of different size limits on the performance of CGMM in next Section. The discovery of new frequent patterns from the *Frequent Pattern Set List* involves the following steps; note that Steps 2 and 3, which comprise the most computational intensive phases of the program, are being executed using GPU:

1. Extract a group of *Frequent Pattern Sets* from the list
2. Generate frequent pattern candidates using *Frequent Pattern Sets*
3. Compute the *supports* of candidates using *Frequent Pattern Sets* and *Weight Vectors*
4. Identify new frequent patterns from the candidates.
5. Add new *Frequent Pattern Sets* created from the new frequent patterns and repeat step 1 until no more *Frequent Pattern Set* is found from the list.

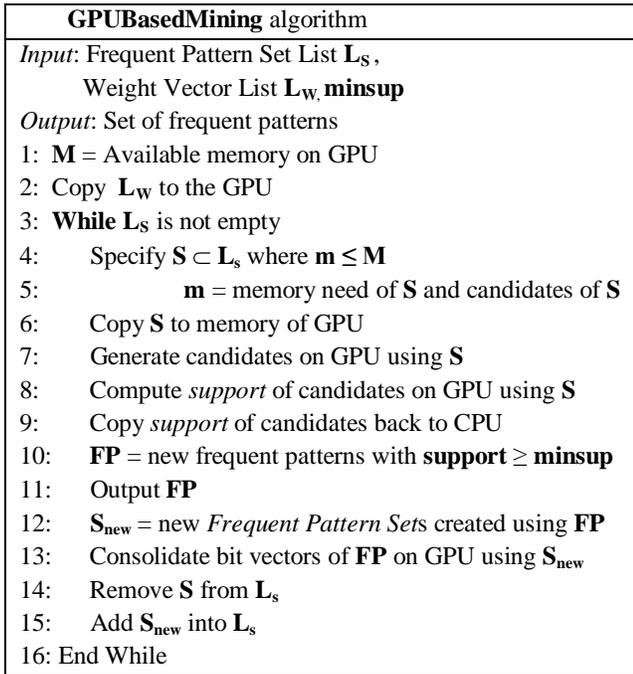


Figure 7: GPUBasedMining algorithm

GPUBasedMining processes a group of *Frequent Pattern Sets* at a time by extracting multiple *Frequent Pattern Sets* from the list as long as their total memory size of newly generated candidates, their bit vectors and *Frequent Pattern Sets* used to generate them do not exceed the available memory on GPU. This workload computation helps CGMM flexibly scale to the memory size of physical device which is a major challenge in GPU computing. In addition, applying the data list structure allows GPUBasedMining working without recursion (recursive procedures do not generally yield high performance on GPUs). Figure 7 presents the algorithmic description of

GPUBasedMining. In this figure, the computation steps involving the GPU include lines 1, 6 – 9 and 13 respectively and are detailed in the following section.

Generating Frequent Pattern Candidates and Computing Their Supports on GPU

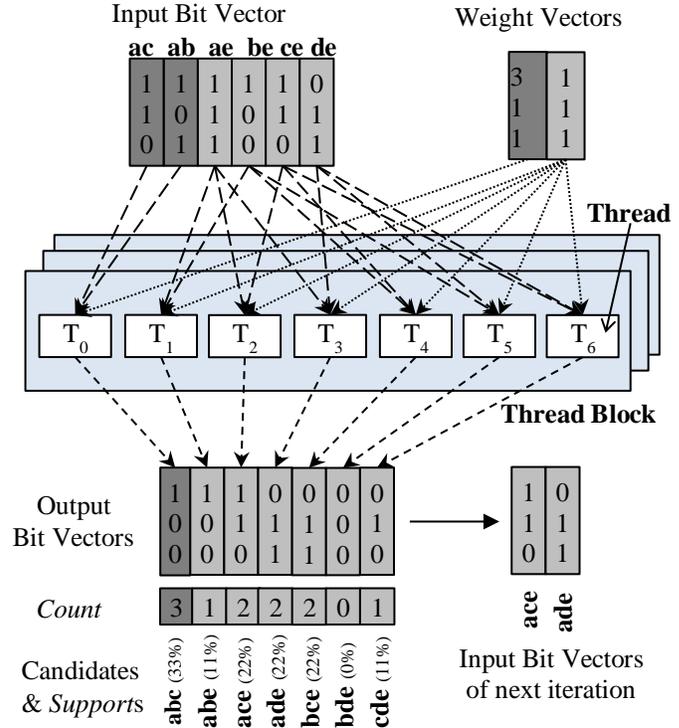


Figure 8: Generating pattern candidates and computing their counts on GPU

Unlike the related works that use GPU for the *support* counting phase only, GPUBasedMining performs both the frequent pattern candidate generation (step 2) and *support* counting (step 3) phases on GPU to reduce the computation handled by CPU; it better utilizes the GPU and therefore improves the efficiency of GPUBasedMining as well as the overall performance of CGMM. The *Frequent Pattern Sets* and the *Weight Vectors* are copied to GPU and distributed among the *thread blocks*. Each concurrent thread in a *thread block*, based on the information of *Frequent Pattern Sets* assigned to its block, will identify the candidates it needs to work on and specify all the necessary information of these candidates such as the two parent frequent patterns, their input bit vectors, its output bit vector to store the result of ANDing the parent's bit vectors. Candidates are then generated in parallel by concurrent threads on GPU and their *supports* are computed. Each thread is responsible for the *support* of one or more candidates independently. Figure 8 illustrates the computations on GPU to generate the frequent pattern candidates and computation of the *supports* using the *Frequent Pattern Sets* and *Weight Vectors* in Figure 6.

Data Transfer Optimization: an important feature of GPUBasedMining is that the output bit vectors of bitwise operation on GPU are not copied back to the main memory. Instead, the bit vectors belonging to the new frequent patterns are consolidated and used as the inputs in the next iteration. For this reason, the new *Frequent Pattern Sets* (S_{new} in line 12 of Figure 7) which are stored in main memory do not include bit vector data. This technique minimizes the communication cost between CPU and GPU, saves memory on the CPU side and enhances the performance of CGMM. For example, for $minsup=20\%$, the new frequent patterns are *abc, ace, ade, bce* because their *supports* $> 20\%$. Among those, *abc, ade* are used to create a new *Frequent Pattern Set* to add to the *Frequent Pattern Set List* because they can be used to create the candidates *acde* in the next iteration. Therefore, the bit vectors of *abc, ade* are kept and unified in the memory of GPU.

PERFORMANCE EVALUATION

Experimental Setup

Datasets: They represent various characteristics and domains of interest for our experiments: three sparse, one moderate and two dense databases all obtained from the FIMI Repository [32], a well-known repository for FPM. The database features are reported in Table 2

Table 2: Experimental datasets of CGMM

Dataset	Type	# of Items	Average Length	# of Trans.
Chess	Dense	76	37	3196
Pumsb	Dense	2113	74	49046
Accidents	Moderate	468	33.8	340183
Retail	Sparse	16470	10.3	88126
Kosarak	Sparse	41271	8.1	990002
Webdocs	Sparse	52676657	177.2	1623346

Software: CGMM can be implemented using different programming platforms like CUDA [3] or OpenCL [4]. In our experiments, we choose CUDA to implement CGMM because CUDA delivers better performance for the Nvidia GPUs used in our experiments. We have carefully tested

our implementation and have verified that it generates correct outputs in every case; this is often a challenge for complex applications developed on GPUs.

Hardware: We use an Altus 1702 machine with dual AMD Opteron 2427 processor, 2.2GHz, 24GB memory and 160 GB hard drive. This machine is equipped with NVIDIA Tesla Fermi C2050 GPU that has 3GB memory, 14 multiprocessors and each multiprocessor consists of 32 CUDA cores 1.5GHz. The operating system is CentOS 5.3, a Linux-based distribution.

Performance Evaluation

To evaluate performance of CGMM, we benchmarked it with six state-of-the-art FPM algorithms Apriori [5], Eclat [23], FP-growth [24], FP-growth* [27], AIM2 [33] and DFEM [26]. Unlike CGMM with multi-strategy approach using both CPU (sequential execution) and GPU (parallel execution), these algorithms apply only one mining strategy and use CPU as the only computing engine. The running time of the seven methods on six datasets with various *minsup*s are given in Figure 9. The experimental results show that CGMM outperforms the other algorithms including Apriori, Eclat, FP-growth, FP-growth* and AIM2 on both dense and sparse datasets for most test cases. Please note that the y-axis of the graphs is in logarithmic scale. CGMM does not run as well as DFEM for larger *minsup* values but it outperforms DFEM when *minsup* reduces. Hence, we recommend to apply CGMM for applications that uses low *minsup* values. For test cases with low *minsup* values, CGMM runs 1.0 – 229 times faster five compared algorithms except DFEM on test datasets (Table 3). It runs faster than DFEM 1.0 – 1.8 times on Chess, Pumsb, Accidents, Kosarak and Webdocs datasets. For Retail, DFEM performs better than CGMM. However, their time difference reduces as *minsup* is set to smaller values. When *minsup* is set to smaller values, the number of data subsets mined by *GPUBasedMining* of CGMM is large and GPU is more efficiently utilized. When *minsup* is larger, the amount of work delegated to GPU is small and this device is under-utilized. In such cases, the highly optimized DFEM is a better FPM solution.

Table 3: Speedup of CGMM vs. other sequential algorithms

Datasets	Minsup	vs. Apriori	vs. Eclat	vs. FP-growth	vs. FP-growth*	vs. AIM2	vs. DFEM
Chess	20%	78.4	3.1	3.5	19.2	1.9	1.1
Pumsb	50%	229.2	2.0	2.5	13.3	5.7	1.3
Accidents	3%	N/A	1.3	1.6	6.8	7.2	1.0
Retail	0.003%	3.7	12.2	2.0	7.8	11.7	0.7
Kosarak	0.08%	15.8	14.6	6.7	12.3	1.0	1.8
Webdocs	4%	45.5	1.1	1.3	2.4	3.3	1.1

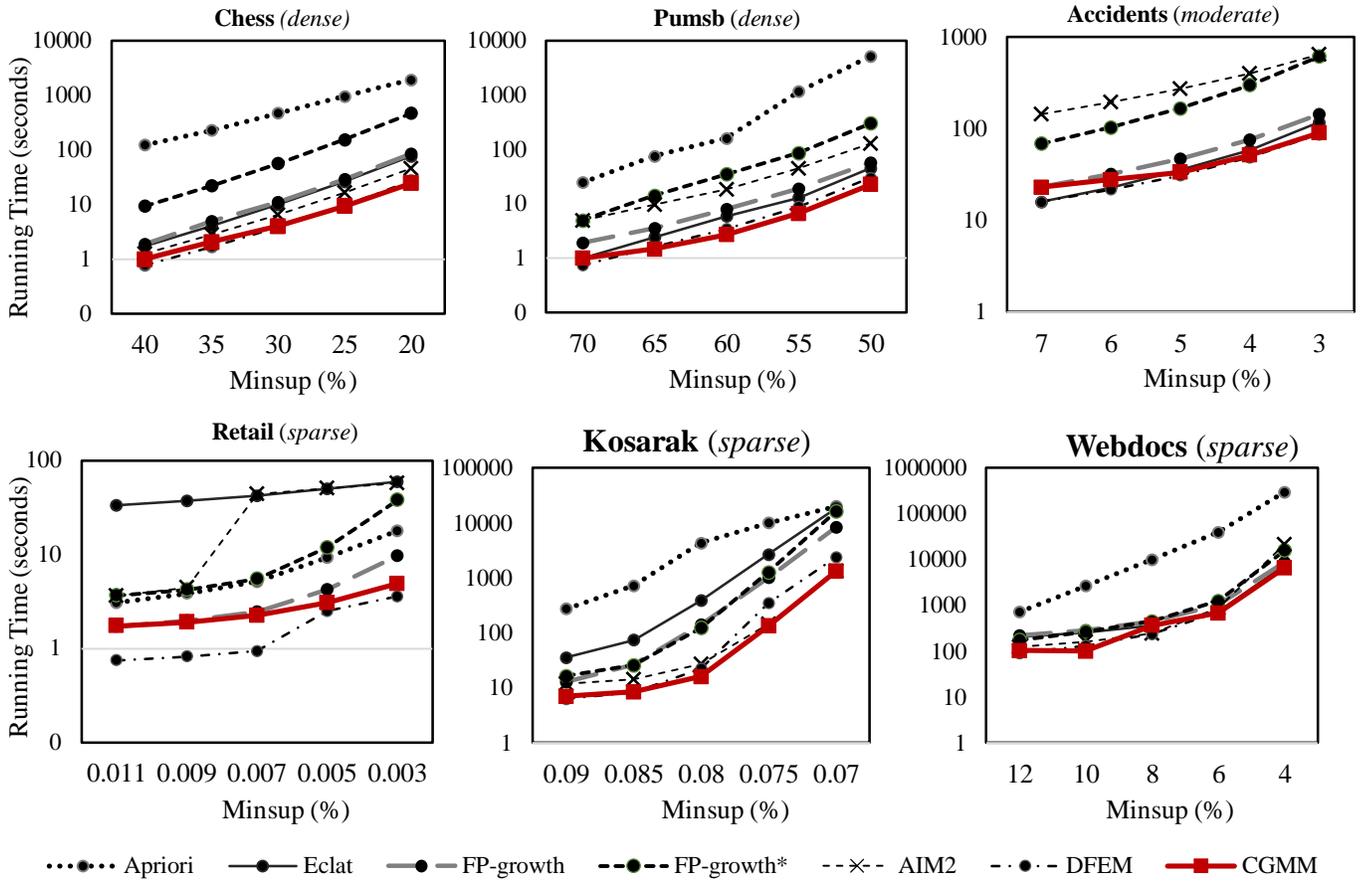


Figure 9: Running Time of CGMM vs. other sequential algorithms

We compare CGMM with GApriori, a GPU based FPM algorithm [20]. Figure 10 shows CGMM runs 7.2-13.9 times faster than GApriori on Retail dataset. In this test case, GApriori uses GPU for entire dataset while CGMM uses GPU for only mining dense data subsets. GApriori failed to run on other datasets because of the internal errors of this program.

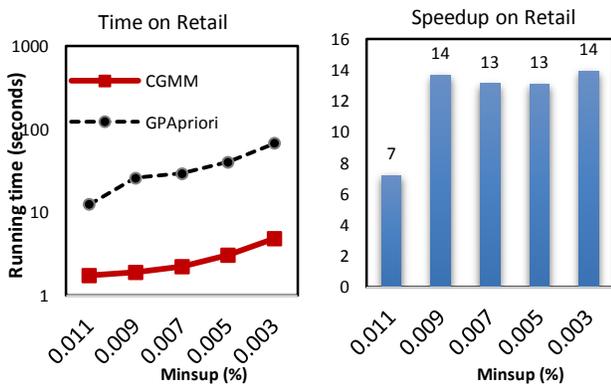


Figure 10: Time and speedup of CGMM vs. GApriori

Upon the results on Retail, we find that for FPM problem, benefits of GPU is only obtained when we use it with suitable data structure and mining solution that can best

leverage the computing power of GPU and adapt well to its limitation of memory and large data communication (e.g. mining dense data subset and low *minsups* values).

Impact of Using Multi-Strategy Approach in CGMM

To study the benefits of applying the two mining strategies, we measured the time of CGMM in three separate cases: (1) using CPUBasedMining only, (2) using GPUBasedMining only and (3) using the combination of CPUBasedMining and GPUBasedMining as intended. The experimental results in Table 4 show that combining the two mining strategies significantly reduces the running times for both sparse and dense databases compared to the cases where only one of the mining strategies were used. For example, CGMM with only CPUBasedMining took 1160 seconds to mine the Chess dataset while CGMM with both strategies ran in only 107 seconds which is 10.8 times faster. Similarly, for Accidents dataset, CGMM with both strategies performed 39.4 times faster than CGMM with only GPUBasedMining (i.e. 419 seconds vs. 16503 seconds). This performance gain comes from the ability to select the suitable strategy for each subset of data being mined to optimize the mining performance.

Additionally, we find that utilizing GPU for FPM does not always bring better performance for all types of data. Although the computing throughput of GPU is hundred

times larger than one CPU, using GPU can sometimes downgrade the overall performance of a FPM task compared to using CPU because of GPU's memory limitations. It explains why for Accidents and Webdocs with very large memory requirements, running only GPUBasedMining performs much slower than running only CPUBasedMining. Combining the two and dynamic switching between them however results in improved overall performance.

Table 4: Running time in seconds of CGMM using two mining strategy vs. using only one strategy

Dataset	CPUBasedMining	GPUBasedMining	CGMM
Chess	1160	180	107
Pumsb	2310	664	377
Accidents	547	16503	419
Retail	226	19	18
Kosarak	2773	841	680
Webdocs	6736	56017	6496

CONCLUSION

We have presented CGMM, a new CPU-GPU hybrid method for FPM. CGMM uses GPUBasedMining strategy for dense data subsets of database and CPUBasedMining for sparse ones. Our experimental results show that CGMM runs up to 229 times faster than six sequential algorithms on six real datasets and 7.2-13.9 times faster than GPAPriori, a GPU based algorithm for FPM. Additionally, CGMM has the ability to self-balance the workload between its two mining strategies based on the characteristics of the database to execute significantly faster than using one mining strategy.

REFERENCES

1. T. Morgan. Top 500 supersedes the dawn of the GPUs. http://www.theregister.co.uk/2010/05/31/top_500_supers_jun2010/, May 2010.
2. Hou, R., Jiang, T., Zhang, L., Qi, P., Dong, J., Wang, H., Gu, X. and Zhang, S. Cost effective data center servers. In *Proc. 2013 IEEE 19th International Symposium on High Performance Computer Architecture* (2013), 179-187.
3. Nvidia. CUDA Programming. in *Best practices guide*, <http://www.nvidia.com/cuda>, 2013.
4. Munshi, A. OpenCL 1.0 Specification. in *Khronos OpenCL Working Group*, 2008.
5. Agrawal, R. and Srikant, R. Fast Algorithms For Mining Association Rules In Large Databases. In *Proc. 20th International Conference on Very Large Data Bases* (1994), 487-499.
6. Brin, S., Motwani, R. and Silverstein, C. Beyond Market Baskets: Generalizing Association Rules To Correlations. In *Proc. the 1997 ACM SIGMOD international conference on Management of data* (1997), 265-276.
7. Silverstein, C., Brin, S., Motwani, R. and Ullman, J. Scalable Techniques for Mining Causal Structures. *Data Mining and Knowledge Discovery* (2000), vol. 4, no. 2-3, 163-192.
8. Agrawal, R. and Srikant, R. Mining Sequential Patterns. In *Proc. the Eleventh International Conference on Data Engineering* (1995), 3-14.
9. Mannila, H., Toivonen, H. and Verkamo, A. Inkeri. Discovery of Frequent Episodes in Event Sequences. *Data Mining and Knowledge Discovery* (1997), vol. 1, no. 3, 259-289.
10. Han, J., Dong, G. and Yin, Y. Efficient Mining of Partial Periodic Patterns in Time Series Database. In *Proc. the 15th International Conference on Data Engineering* (1999), 106-115.
11. Han, J., Cheng, H., Xin, D. and Yan, X. Frequent Pattern Mining: Current Status And Future Directions. *Data Mining and Knowledge Discovery* (2007), vol. 15, no. 1, 55-86.
12. Burdick, D., Calimlim, M., Flannick, J., Gehrke, J. and Yiu, T. MAFIA: A Maximal Frequent Itemset Algorithm. *IEEE Transactions on Knowledge and Data Engineering* (2005), vol. 17, no. 11, 1490-1504.
13. Cui, Q. and Guo, X. Research on Parallel Association Rules Mining on GPU. In *Proc. the 2nd International Conference on Green Communications and Networks 2012* (2010), vol. 224, pp. 215-222, 2010.
14. Teodoro, G., Mariano, N., Jr., W. M. and Ferreira, R. Tree Projection-Based Frequent Itemset Mining on Multicore CPUs and GPUs. in *Proceedings of the Symposium on Computer Architecture and High Performance Computing* (2010), 47-54.
15. Jian, L., Wang, C., Liu, Y., Liang, S., Yi, W. and Shi, Y. Parallel data mining techniques on Graphics Processing Unit with Compute Unified Device Architecture (CUDA). *Supercomputing* (2013), vol. 64, 942-967.
16. Kozawa, Y., Amagasa, T. and Kitagawa, H. Parallel and Distributed Mining of Probabilistic Frequent Itemsets Using Multiple GPUs. *Lecture Notes in Computer Science, Database and Expert Systems Applications* (2013), vol. 8055, 145-152.
17. Lin, C.-Y., Yu, K.-M., Ouyang, W. and Zhou, J. An OpenCL Candidate Slicing Frequent Pattern Mining Algorithm on Graphic Processing Units. in *Proceedings of the 2011 IEEE International Conference on Systems, Man, and Cybernetics* (2011), 2344- 2349.
18. Silvestri, C. and Orlando, S. gpuDCI: Exploiting GPUs in Frequent Itemset Mining. in *Proceedings of 2012 20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing* (2012), 416-225.
19. Huang, Y.-S., Yu, K.-M., Zhou, L.-W., Hsu, C.-H. and

- Liu, S.-H. Accelerating Parallel Frequent Itemset Mining on Graphics Processors with Sorting. *Lecture Notes in Computer Science, Network and Parallel Computing* (2013), vol. 8147, 245-256.
20. Zhang, F., Zhang, Y. and Bakos, J. GPAPriori: GPU-Accelerated Frequent Itemset Mining. in *Proc. the 2011 IEEE International Conference on Cluster Computing* (2011), 590-594.
 21. Kozawa, Y., Amagasa, T. and Kitagawa, H. Fast Frequent Itemset Mining from Uncertain Databases using GPGPU. in *Proc. the Fifth International VLDB Workshop on Management of Uncertain Data* (2011), 892-901.
 22. Zhou, J., Yu, K.-M. and Wu, B.-C. Parallel Frequent Patterns Mining Algorithm on GPU. in *Proc. the 2010 IEEE International Conference on Systems Man and Cybernetics* (2010), 435-440.
 23. Zaki, M., Parthasarathy, S., Ogihara, M. and Li, W. New Algorithms for Fast Discovery of Association Rules. In *Proc. the 3rd International conference on Knowledge Discovery and Data Mining*, pp. 283-286, 1997.
 24. Han, J., Pei, J. and Yin, Y. Mining Frequent Patterns Without Candidate Generation. In *Proc. the 2000 ACM SIGMOD international conference on Management of data*, New York, NY, USA, pp.1-12, 2000.
 25. Vu, L. and Alaghband, G. A Fast Algorithm Combining FP-Tree and TID-List for Frequent Pattern Mining. In *Proc. the 2011 International Conference on Information and Knowledge Engineering* (2011), 472-477.
 26. Vu, L. and Alaghband, G. Mining Frequent Patterns Based on Data Characteristics. In *Proc. the 2012 International Conference on Information and Knowledge Engineering* (2012), 369-375.
 27. Grahne, G. and Zhu, J. Fast Algorithms for Frequent Itemset Mining Using FP-Trees. *IEEE Transactions on Knowledge and Data Engineering* (2005), vol. 17, issue 10, 1347-1362.
 28. Pei, J., Han, J., Lu, H., Nishio, S., Tang, S. and Yang, D. H-Mine: Hyper-Structure Mining of Frequent Patterns in Large Databases. In *Proc. the IEEE International Conference on Data Mining* (2001), 441-448.
 29. Racz, B. nonordfp: An FP-Growth Variation without Rebuilding the FP-Tree. In *Proc. the 2004 Workshop on Frequent Pattern Mining Implementations* (2004).
 30. Liu, L., Li, E., Zhang, Y. and Tang, Z. Optimization of Frequent Itemset Mining on Multiple-Core Processor. In *Proc. the 33rd international conference on Very large data bases* (2007), 1275-1285.
 31. Moriwal, R. FP-growth Tree for large and Dynamic Data Set and Improve Efficiency. *Information and Computing Science* (2014), vol. 9, no. 2, 559-583.
 32. Frequent Itemset Mining Implementations Repository. *Workshop on Frequent Itemset Mining Implementation*, 2003-2004.
 33. Shporer, S.. AIM2: Improved Implementation of AIM. In *Proc 2004 Workshop on Frequent Itemset Mining Implementations* (2004).
 34. Schmidt-Thieme, L. Algorithmic Features of Eclat. In *Proc. 2004 Workshop on Frequent Itemset Mining Implementations* (2004).
 35. Orlando, S., Lucchese, C., Palmerini, P., Perego, R. and Silvestri, F. kDCI: a Multi-Strategy Algorithm for Mining Frequent Sets. In *Proc. 2003 Workshop on Frequent Itemset Mining Implementations* (2003).
 36. Fiat, A., and Shporer, S. AIM: Another Itemset Miner. In *Proc. 2003 Workshop on Frequent Itemset Mining Implementations* (2003).

Computational Steering for High Performance Computing

Applications on Blue Gene/Q System

Bob K. Danani

IBM Research Australia
Level 5, 204 Lygon St, Carlton
VIC 3053, Australia
bob.danani@au1.ibm.com

Bruce D. D'Amora

IBM T. J. Watson Research Center
101 Kitchawan Rd 134, Yorktown Heights
NY 10598, USA
damora@us.ibm.com

ABSTRACT

The traditional workflow in a high performance computing (HPC) simulation is to prepare the application's input, run the simulation, and visualize the simulation results in a post-processing step. By performing these steps simultaneously, significant development and testing time can be saved. Computational steering provides the capability to direct or re-direct the progress of an HPC application at run-time by modifying application-defined control parameters using a steering client application. In this paper, we discuss a computational steering framework for the Blue Gene/Q system that provides an innovative solution and an easy-to-use platform, which allows user(s) to connect to and interact with running application(s) in real-time from native desktop steering applications and/or mobile devices. This framework uses RealityGrid as the underlying steering library and adds several enhancements to the library to enable steering support for the Blue Gene systems. The Blue Gene supercomputer presents special challenges for remote access because the compute nodes reside on private networks. This paper discusses an implemented solution for remote steering of simulation applications running on a high performance computer system and describes the implementation challenges.

Keywords

Computational Steering; Visualization; Blue Gene/Q;
Runtime System; HPC Workflow

INTRODUCTION

The evolution of High Performance Computing (HPC) has been one of the most important trends in the computing world over the last few decades. Supercomputers have evolved from simple machines with only a few processors to massively parallel machines utilizing more than a million cores. Many complex computing applications, such as simulation and physical modeling applications, behavioral modeling applications, and real-world data acquisition applications, require machines that have very high compute power capabilities, very large bandwidth, and very low network latency. These needs have resulted in a demand for an optimal HPC workflow.

The most common computational workflow in an HPC simulation is to prepare the application's input, run the simulation, and visualize the simulation results in a post-processing step. In many scientific applications there often exists a need to test multiple control parameter configurations in order to pick the most optimal parameter configuration, such as a parameter configuration that yields the best performance, produces the most accurate prediction, or greatly reduces the computation time of a simulation. These sequence of steps need to be repeated every time the corresponding application needs to be tested with different control parameter settings and this impacts time to solution.

In various HPC simulation applications, there is also a need to send data between multiple applications. One common case is to couple simulation results with a visualization module or other downstream analytics applications. Typically, visualization is done offline as a separate process from the simulation. It is also common to observe a lot of data movement to and/or from different storage systems.

Figure 1 illustrates the typical computational workflow in HPC applications. In this workflow, simulation, data integration, and data analytics are conducted sequentially rather than in parallel. In this figure we have several components of a typical workflow. First, there are three types of modeling – physically based simulation modeling, behavioral modeling, and real world data acquisition

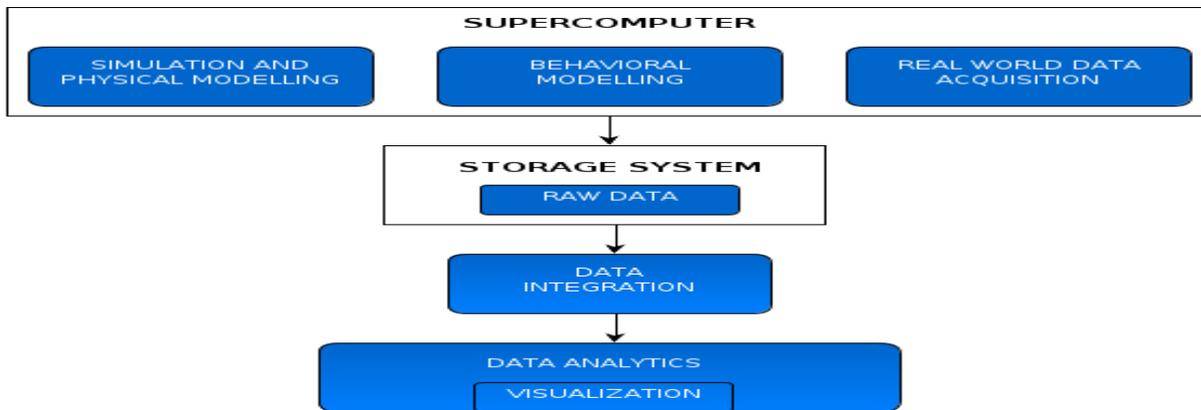


Figure 1: Typical HPC Steering Workflow [1]

modeling. Next, there are control parameters that are set for the application. These parameters can control the length of the time step, the behavior at the model boundaries or the quality of the materials and other properties in the simulation. In addition to control parameters, the workflow includes a raw data output stage that is implemented by writing to files. In the case of simulation applications, the amount of data may be very large. It is common to collect data at specific time intervals, i.e., through checkpointing, or only at the last time step. Since on very large systems there may be several component applications running in concert, e.g. Nuclear Reactor simulation or Combustion simulation, there must be a data integration step in the workflow. Once applications are running and data is being collected, analytics can then be performed. This workflow typically requires the user to query for files that are being written to disk, move data to another system for data analysis or visualization, and explicitly change the control parameters for the simulation via a file edit, which overall is tedious and susceptible to human error. This scenario makes the current workflow slow and inflexible for heavyweight HPC simulation applications, which implies that a better HPC computational workflow is needed.

Computational steering is important in many areas of HPC because it allows real-time interactivity to direct or re-direct the progress of a simulation application at runtime. It enables users to modify application-defined control parameters using various user-steering applications. Computational steering enables a transition from a typical batch computational mode into a more interactive computational mode. It improves productivity by greatly reducing the latency time between the modification to parameter values and the viewing of the results [2]. It can be used to adjust control parameters of algorithmic applications that contain some behaviors that are hard to understand and are usually dependent on the input data characteristics [3]. Additionally, it also offers the ability to examine a simulation’s behavior to gain additional insight of it [4].

In this paper, we discuss a computational steering framework for the Blue Gene/Q system that provides an innovative solution and an easy-to-use platform, which allows user(s) to connect to and interact with running application(s) in real-time from native desktop steering applications and/or mobile devices. This framework uses RealityGrid [5] as the underlying steering library. We propose and implement an additional design point for the RealityGrid computational steering framework to support the Blue Gene/Q architecture, which was designed by IBM to deliver petascale-computing performance. We provide an innovative solution and an interface for the user to modify runtime parameters in a running computer simulation. The Blue Gene/Q supercomputer presents special challenges for remote access due to its architectural design because its compute nodes reside on a private network, which is inaccessible to other systems on intranet or internet. This paper discusses an implemented solution to it and discusses additional challenges for remote steering of simulation applications running on a high performance computer system.

MOTIVATION

Serial execution of a simulation application, data integration, and data analytics leads to a longer time to solution and unavoidable machine time costs, which prohibit excessive amount of *trial and error* to converge to the optimal results. A solution to this is to provide feedback to the simulation based on its intermediate results, by having the application examine the intermediate output results obtained at a particular time step. Unfortunately, this approach appears to be tedious and non-productive in determining what is exactly going on at each intermediate step, mainly because of the high volume of data that is typically generated by the simulation applications. Large-scale scientific applications enabled by the advancement in HPC technologies have led to continuous simulations that span over thousands of time steps, each having much iteration before a simulation result can be determined. It is not uncommon for a large scale and complex application (i.e., an oil reservoir visualization simulation or a large

protein network simulation) to have each simulation time step generating over thousands to millions of datasets, which overall yields to gigabytes or even terabytes of storage usage. A more direct and practical method is to couple the simulation with a visual analysis of the intermediate results. In this way, the user can investigate the results of the simulation globally and at the same time leverage data analytics to make changes to the running simulation. For example, a user may have a simulation that requires a structured or an unstructured grid model. One element of the simulation problem that can be varied by the user is the boundary conditions. In this case, the user may be able to switch from a fixed boundary to a linear boundary or a time-varying boundary condition. Other control parameters that may be made steerable are the time step and the material property. The goal is to offer runtime user control of these different steerable parameters to effectively steer the progress of the simulation.

In addition to controlling the application with steerable parameters, it is often desirable to obtain a visualization of its intermediate results. For example, a computational fluid dynamics (CFD) application may provide so much data that it is difficult for the user to understand whether or not the simulation is progressing correctly. Visualization can be used to provide a snapshot that is easier for the end-user to process. Accomplishing this while a simulation is running requires a data transfer from the simulation to the visualization application. The visualization application may be running on a supercomputer or it may be down-streamed on a dedicated visualization server. In either case it is important to integrate these applications with a steering library that provides an Application Programming Interface (API) for network data transfer.

A fundamental requirement for computational steering is the capability to move data from an HPC application to different storage devices or memory devices, whether

located locally or remotely to the application. Additionally, computational steering also requires communication between the applications that are being steered and the applications that provide steering capability. These applications can potentially all be executing on the same cluster or can be executing on variety of devices from handhelds to departmental supercomputers and visualization clusters. This scenario leads to a demand for a more robust and optimal workflow, as depicted in Figure 2. This scenario is almost identical to that described in Figure 1 with the exception of a feedback loop added between the user-controlled steering application and the control parameter input into the simulation. It provides a mechanism and a path for the user to modify the control parameters at runtime. We call these control parameters steerable parameters. Steerable parameters may include time step length, fluid properties, material properties, boundary conditions, multi-scale grid resolutions, etc. The steering application may be a plug-in component to an existing application or it can be a stand-alone application.

This future workflow has several important features that revolutionize the previous workflow, which include:

- i) an ability to modify control parameter values in real-time, which means that no simulation restart is required;
- ii) a change in the way raw output data used as input by downstream applications is stored, i.e., switching from intermediary storage systems to data streaming via network;
- iii) parallel execution of simulations, data integration, and analytics (visualization); and
- iv) support for multiple user interaction in a collaborative environment, allowing users to interact with the running simulation applications from a variety of devices, which can range from thin client devices (i.e., laptops, mobile phones, tablets, etc.) to workstations.

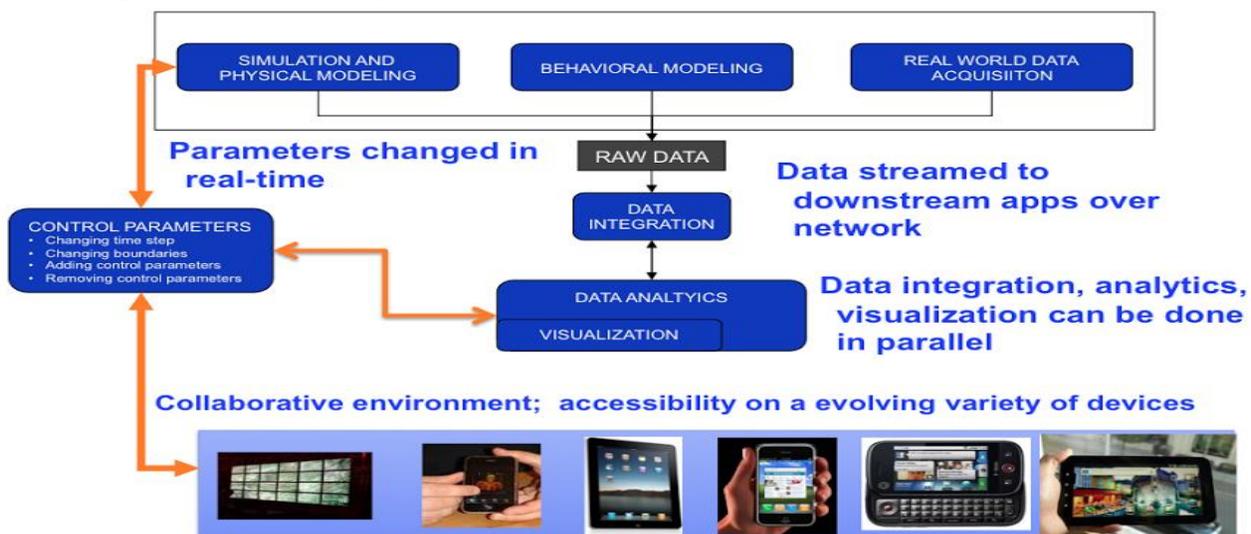


Figure 2: Future Steering Workflow [1]

BLUE GENE/Q PLATFORM

This work has been done on the IBM T.J. Watson Research Center Wat2Q Blue Gene/Q supercomputer. The IBM Wat2Q system is a two-rack HPC system; each containing 1024 nodes and each node has an 18-core PowerEN (TM) chip running at 1.6 GHz, among which 16 of them are used for computing, 1 for operating system services, and 1 as a redundant spare core. Each compute node (CN) has 16 GB DRAM shared by the 16 compute cores. The Input / Output (I/O) on Wat2Q is provided via I/O nodes (ION), which are shared by CN partitions. Each ION is comprised of a 16-core PowerEN (TM) chip running RedHat Linux. Each rack has a space for 4 I/O drawers each holding up to 8 I/O boards, which makes it a total of 32 ION per rack.

REALITYGRID STEERING LIBRARY

RealityGrid [5] is an open source computational steering library developed by the University of Manchester, University College London, University of Edinburgh, Imperial College, University of Oxford, and Loughborough University. The steering library was developed to provide an API for integration with simulation applications, which will enable the modification of control parameters at runtime to control the progress of those simulations. The steering library offers the ability to pause, resume, and stop the execution of a running simulation application. This library supports two types of parameter registration: steerable parameters and monitored parameters. Steerable parameters are parameters whose values can be modified by users during runtime. Monitored parameters are read-only parameters whose values are observed during the execution of an application, but cannot be steered by users. The main goal is to provide a functionality to steer a simulation application, but ultimately the same library can also be used to steer the visualization component or any downstream application, such as a data analytics application.

RealityGrid supports registration of checkpoints. Additionally, the library also provides API to control the emission and consumption of data produced by the simulation application so it can be visualized by other application(s). An API is provided to establish connections with steering and visualization clients using one of these interfaces: (1) files, (2) sockets, and (3) Web Services Resource Framework (WSRF). In all cases, the steering commands between the clients and the simulation application are always embedded in XML streams. Our implementation is derived from the socket-based steering API in RealityGrid steering library. In the original implementation, a steering-client application will need to connect to an already-running simulation application via TCP sockets. Once a socket connection is established, the steering application and the steered application will begin to communicate with each other and exchange steering commands through this direct socket connection.

The RealityGrid steering library was developed for a more general supercomputing architecture rather than the Blue

Gene platform. Due to the restrictive network access in the Blue Gene system, which disallows connections from external networks, it is not possible to run a steering client application on an external system and have it connected to a simulation application that runs inside the Blue Gene CN. Hence, a better steering solution is needed to support the Blue Gene architecture and possibly should also work for other types of architectures.

IMPLEMENTATION ISSUES

The goal of this project is to enable steering of applications that are running on CN from various external steering client devices. Since the client is often on an external network, this becomes problematic. Blue Gene applications run on CN. CN typically runs a very lightweight operating system called the Compute Node Kernel (CNK). CN do not directly establish network connections to the external network. They access the external network via ION. CN have unique network IP addresses assigned to them. The relationship between CN and ION is similar to that of a network gateway router and devices connected to the external network through that gateway. This presents the following complication: the RealityGrid steering library establishes the simulation application as a socket server that requires an external client application to request binding with an open socket. This is currently only possible if the external client application shares the same subnet as the compute nodes. In a typical Blue Gene installation the only servers that share the same private network as CN are the front-end-nodes (FEN), which are used to launch applications, and the service node, which is used to administrate various system control function. The service node is not accessible by the general user.

We have made modifications and enhancements to RealityGrid to support the Blue Gene/Q system. These enhancements are categorized into application side, front end node and relay server side, and steering client side. Applications may include those executing on Blue Gene systems or applications executing on other network connected systems, i.e., visualization applications. The steering clients may be interfaced as web applications or native device applications. Steering clients may also be executing on mobile devices, including but not limited to Android phones, IOS devices (iPhone / iPad), or any similar tablet devices. The relay server acts as a conduit between the steering clients and the applications, marshaling steering commands and parameters back and forth. Commands can be divided into two categories: steering library default function set (i.e., start, stop, pause, resume) and user-defined functions. They control the application flow. Parameters are typically defined by the application and they control the computation results. The different components of the steering framework are described in the next three subsections.

Application Side

One important design view in RealityGrid is that applications should work normally even if they are not integrated with the steering library. Integration with the steering library is done for the purpose of making the applications steerable, but not necessarily modifying the existing features in the applications. RealityGrid provides support for I/O communication between two different applications in which one application can act as a producer and another application as a consumer. I/O communication channels need to be registered during the application's initialization stage.

RealityGrid does not prevent integration with MPI applications. However, this library was not designed to support a full MPI parallelism that allows more than one MPI rank to establish the I/O channels. Usage of RealityGrid on an MPI application assumes that only one MPI process (usually MPI rank0) should establish the I/O channel, either as a producer or as a consumer. Once the communication channels are established, MPI rank0 can then perform various inter-communicator collective MPI operations (i.e., broadcast, scatter, gather, reduce, scan, all-gather, all-reduce, or all-to-all).

Front End Node and Relay Server Side

The Blue Gene/Q system is typically installed with a private network connecting CN and ION and a public intranet connecting FEN and service nodes to external users. FEN maintains a network connection to the private compute node network as well as the external network. For overall system security, the private network is typically inaccessible from any system not on the same subnet so any communication between the simulation application running on the compute nodes and the remote client devices is usually not direct. In this case, we have created a relay server running on the front-end nodes to act as an intermediary between the private compute network and systems on an organizations intranet or the internet.

The relay server is a socket server that can accept connections from applications executing on the Blue Gene compute nodes as well as from remote steering clients that execute on different devices. This approach was chosen because of the limited support for sockets on the CNK operating system, where a socket can only act as a socket client. The relay server manages all of these socket connections, which are implemented either as standard TCP sockets or WebSockets. WebSockets were supported because of the prevalence of web-based clients and the advantages of having a full-duplex connection between the client and the relay server. Once connections are established, the relay server can act as a pass through agent between the running simulations and the remote clients. Furthermore, http clients can be served via a simpler or more elaborate web-services framework such as Apache to remote clients running on PCs or smart devices such as tablets and phones.

Our implementation uses Apache HTTP Server for enabling steering client access from web-browsers. The Apache HTTP Server can be installed on any system as long as there is a connection route from the HTTP server to the relay server that is installed on the FEN. The following sequence takes place whenever a compute node application and a steering client attempt to connect:

- An HTTP server is started to securely serve html pages to remote clients.
- The relay server program is started and it spawns a relay server thread running on FEN. The relay server uses several hash-map data structures to efficiently map the connectivity between the steered application(s) and the steering client(s).
- The compute node application is started. Upon application initialization, it will open a connecting socket to the relay server. The address of the relay server is passed either as an environment variable or as a variable defined in a configuration file. Regardless of how the address is passed, the application needs to know the IP address and the port number of the relay server to be able to connect to it.
- The steering client is started and will request binding with the relay server's open socket. Once this handshake occurs, connection is established and the relay server can then pass through commands from compute node application to steering client and vice versa.

The relay server was initially as simple as previously discussed, but it has evolved into something more complex and will continue to evolve. Since the Blue Gene/Q front-end nodes are IBM Power based systems and steering clients often execute on Intel based system, there needs to be a switch between Little Endian and Big Endian, which requires (1) Endianess determination and (2) byte swapping functionality in the relay server. Additionally, there needs to be some support for registering simulation applications so clients can query a list of currently-running steerable applications and request connections. The relay server is designed to support connections from multiple simulation applications; thus it is multi-threaded. When a simulation application runs, it establishes a socket connection to the main relay server, which has already created a listening socket. The main relay server then instantiates a relay object for each connected application, running in its own thread. Thus, we have a relay running in a thread that clients can connect to send/receive parameters or commands to/from the corresponding application. The steering client can then initiate a socket connection to the relay server thread and begin communicating with the corresponding application, through this relay connection.

A separate lookup server and a simple database server were also developed for the purpose of registering user session, processing queries from steering clients, querying a list of connected users, querying a list of applications that a user currently connects to, and querying other connected users that are connected to a same application.

Steering Client Side

The RealityGrid steering library implements a set of functions for establishing a connection and transmitting and receiving data to/from the simulation application. Basic examples included in the RealityGrid distribution include a generic Qt steering client. This generic steering client provides an API for a client to bind to a listening socket previously opened by the compute node application. Since the client is often on an external network, this is problematic. RealityGrid's socket-based steering approach is not directly usable on the Blue Gene systems without some major modifications as the Blue Gene compute nodes cannot create a listening socket and can only create a connecting socket.

In our modified computational steering framework, the client application needs to initiate a socket connection to the relay server. Some of the new features that are supported in our client side steering framework include, but not limited to:

- an ability to view an updated list of all steerable applications that are currently running and connected to the relay server;
- a feature that enables multiple users to connect to multiple applications and steer these applications simultaneously;
- an ability to view a list of other connected users that are steering the same application;
- an ability to control the I/O emission and/or consumption frequencies in multiple simulation applications; and

- a security enhancement to restrict steering activity only to authenticated users.

To interface with the modified computational steering framework, we developed a C++ Qt desktop client and an HTML5 web-based browser client. The client-server communication in the desktop steering client is managed through standard TCP sockets. The steering client application can have multiple steering views (i.e., one for each connected application) to simplify the management of interactions with each steered application. Multithreading approach is used to manage the socket communication between the steering client application and all the connected applications, i.e., one thread per application. Figure 3 shows the interface design of the desktop steering client.

The web-based browser client is developed based on popular web standards such as HTML5, JavaScript, and CSS. This web-client application interfaces with our steering framework and simulation application via WebSocket. The integration with WebSocket technology will make the simulation applications more accessible from a variety of compute clients, e.g. handhelds, laptops, etc. We also integrate our web-based steering client with Apache Cordova / PhoneGap [6] framework to enable users to interact with simulation applications from various mobile platforms, such as Android, IOS, Blackberry, etc. PhoneGap is an HTML5 application framework that enables developers to develop native mobile applications with web technologies that they already know the best: HTML and JavaScript.

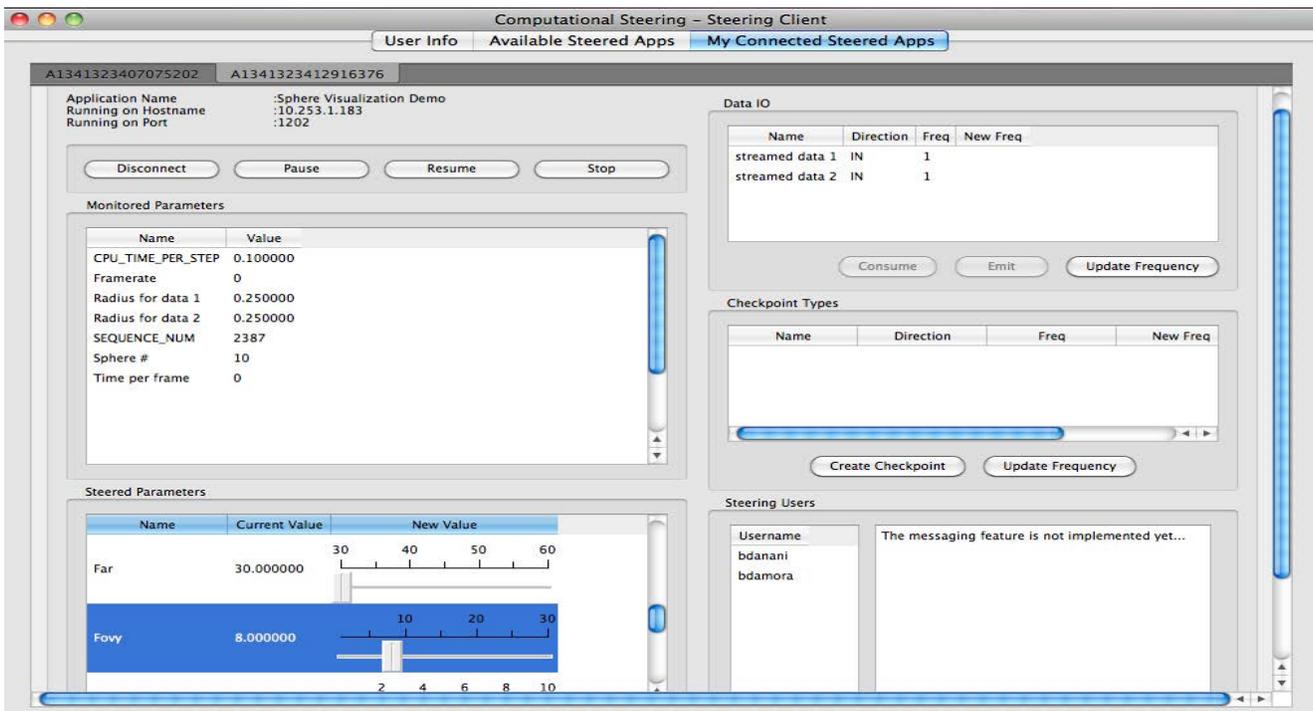


Figure 3: Desktop Steering Client Interface

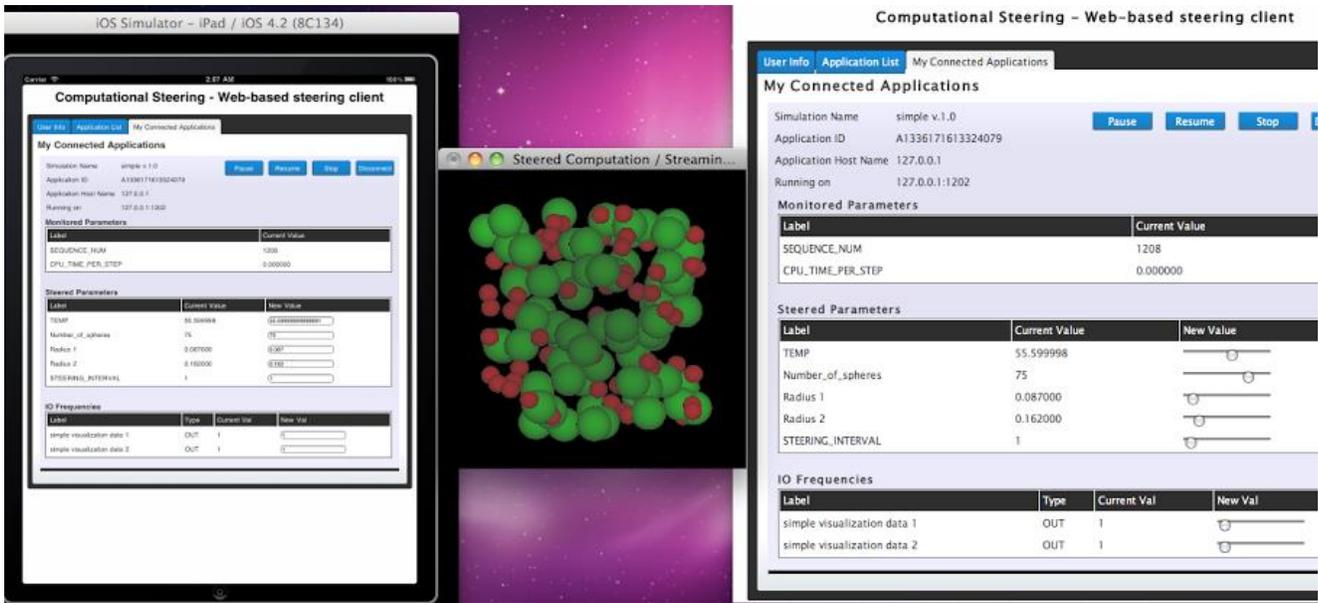


Figure 4: Collaborative User Steering Using Web-based Steering Clients

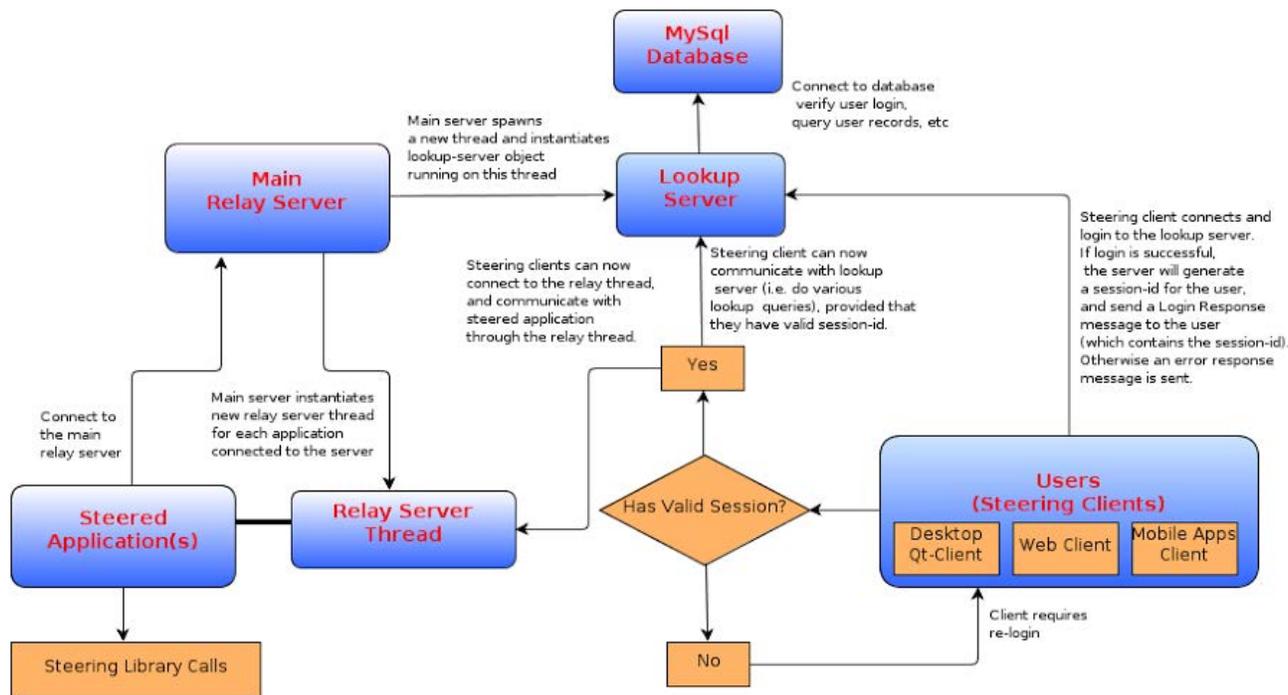


Figure 5: Proposed Computational Steering Architecture

Figure 4 shows a scenario where we have two users interacting simultaneously with a running application using the web-based steering clients; one running on a web browser and another one on an iPad device simulator (IOS 4.2). The IOS client is developed using the PhoneGap framework provided by Cordova. Figure 5 summarizes the architecture design of the relay server and its relation with the other components in this computational steering framework.

APPLICATION TEST CASE

To showcase our work, we used OpenFOAM (Open Field Operation and Manipulation) [7] as our test application. OpenFOAM is a computational fluid dynamics (CFD) toolbox that contains a suite of solvers that use numerical methods to analyze and solve problems that involve fluid flows. Specifically we use a solver called icoFoam as our test case, which is a transient solver for incompressible,

laminar flow of Newtonian fluids. To show the ability of our steering framework to steer multiple parameters and to simultaneously visualize the intermediate result, we modified the codes of icoFoam solver to steer two control parameters: the size of the time step ΔT and the number of the corrections n_{Corr} used in the inner loop. Moreover, we emitted the pressure field mesh to visualize and monitor the intermediate changes of the application.

In this computational steering framework, I/O communication between two or more applications is internally managed by the RealityGrid's I/O Channel mechanism. I/O data is encapsulated in XML format before it is sent over the network. To verify whether or not the XML encapsulation adds a lot of overhead that may impact the application's performance, we developed a simple performance client application and a performance server application and enabled steering in them. These applications communicate with each other by sending and receiving packets of data in a *ping pong* fashion. We calculated the performance bandwidth between the client and server applications and compared them with the ideal bandwidth that could be achieved by the same applications when they are not instrumented with the steering framework. We ran the client application and the server application on separate machines that are connected through different network architectures. We observed that the overhead caused by the XML encapsulation is very small and can be negligible. On a 1 Gbps network interface the overhead is below 1.0%. On a 10 Gbps network interface the overhead is in the range of 0.1% to 0.5%.

CONCLUSION

We designed a computational steering framework for HPC applications running on the Blue Gene/Q system. We designed the functional requirements and built up a prototype based on RealityGrid's socket-based steering API and added various enhancements in its functionalities. Due to the architectural design of the Blue Gene supercomputer, the RealityGrid's socket-based steering design had to be majorly refactored to support this architecture.

By introducing a relay server running on the FEN, we are able to maintain the communication between the steering applications and the HPC applications. We added support for many-to-many connectivity between multiple steered applications and multiple steering clients to encourage multi-user collaborative steering experience.

We developed various steering client applications to interface with the computational steering framework. These steering clients include a desktop steering application, a web-based steering application, and a mobile steering application. By using the PhoneGap framework, we were able to deploy the web-based steering client on multiple mobile devices platforms to improve the user's steering experience. We presented an overview of the architectural

design of our computational steering framework. It is important to note that our computational steering framework can also be readily used for other platforms.

FUTURE WORK

In the future we would like to have a direct connection between the HPC simulation and the visualization application to avoid any data copy at the relay server. We also would like to develop a Paraview [8] plug-in on the Blue Gene/Q CN to provide a better support for visualization and steering using the Paraview CoProcessing library. The CoProcessor will provide support for visualization on Blue Gene/Q, where either the entire visualization or part of the visualization is done on Blue Gene/Q, and the result can then be transferred via sockets to the Paraview server running on another system that can do the rendering to produce the final image for visualization.

The relay server is currently not designed to be aware of multiple points of control, so it will treat each point of control as a new application that registers with the relay server. This is something that we also plan to add in our future work as it is very common for some complex applications to have multiple points of control, such as in multi-physics CFD applications.

ACKNOWLEDGMENTS

This work was supported by a joint collaboration between IBM T. J. Watson Research Center and King Abdullah University of Science and Technology (KAUST). We would like to thank D. Kaushik of Supercomputer Laboratory at KAUST, and J. Sexton of IBM T. J. Watson Research Center who led this collaboration

REFERENCES

1. D'Amora, B.D. Computational Steering of HPC Applications on Blue Gene System. *KAUST HPC Workshop* (2012).
2. Marshall, R. et al. Visualization Methods and Simulation Steering for 3D Turbulence Model of Lake Erie. In *Proc. I3D 1990*, Interactive 3D Graphics Conf., ACM SIGGRAPH Computer Graphics (1990), 89-97.
3. Miller, D.W. An Optimistic Approach to Computational Steering. *M.S. Thesis*, Univ. of Georgia (2002).
4. Mulder, J.D., Wijk, J.V., and Liere, R.V. A Survey of Computational Steering Environments. *Future Generation Computer Systems*, 15 (1999), 119-129.
5. Univ. of Manchester. *RealityGrid*. <http://www.realitygrid.org>.
6. Adobe Systems Inc., *PhoneGap*. <http://www.phonegap.com>.
7. S.G.I. Corp., *OpenFOAM*. <http://www.openfoam.org>.
8. Kitware Inc., *ParaView*. <http://www.paraview.org>

Strategies to Hide Communication for a Classical Molecular Dynamics Proxy Application

Issakar Ngatang, Masha Sosonkina
Department of Modeling, Simulation
and Visualization Engineering
Old Dominion University, Norfolk, VA 23529
Email: {ingat001, msosonki}@odu.edu

ABSTRACT

Co-designing applications and computer architectures has become of major importance due to the growing complexity of both applications and architectures and the need to better match application characteristics to the available hardware. Thus, “mini-applications”, which serve as proxies of large-scale ones by highlighting their most intensive parts and major workflow components, appeared to the co-design, tuning, and adaptation purposes. This paper presents a work on optimizing the communication subsystem of a classical MD proxy (CoMD) application executed on multi-core computing clusters. The research focuses on hiding communication with certain buffer handling operations. In particular, two strategies are presented: one that uses two parallel threads for communication and buffer handling and another that introduces more parallelism by allowing all the available threads to unload the buffers while using two thread to communicate, thereby improving load balancing. The first proposed strategy yields performance gains up to 61% in the communication routines, corresponding to 6% gains in the overall time, while the second strategy achieves, respectively, about 73% and 6.3% improvement.

Author Keywords

Hybrid programming model; OpenMP; MPI; communication/computation overlap; classical molecular dynamics.

ACM Classification Keywords

I.6.1 SIMULATION AND MODELING (e.g. Model Development). : Miscellaneous

INTRODUCTION

Large-scale parallel computing platforms are often heterogeneous featuring various node capacities and several network interconnects in the same system, which increases the system complexity and has a direct effect on the applications running on them, making it difficult to achieve optimal performance. The complexities and scale in both architecture and applications call for their mutual adaptation and tuning to the obtained feedback and specific hardware configuration and application parameters. In [1], Barrett *et al.* discuss potential benefits of software/hardware co-design for future exascale computers. They have established that co-designing software and computer architecture by teams of both application and

hardware designers and sophisticated tools will help dramatically accelerate the design cycle and lead to better performance.

This paper focuses on the communication subsystem of the classical molecular dynamics (CoMD) proxy application and proposes strategies to hide its communication overheads. In the future work, these strategies, having several tunable parameters, will be used to configure the CoMD and other molecular dynamics codes to the interconnection network at hand, non-uniform memory access (NUMA) layout of the node, and to the available MPI implementations. It is more productive to work out and test co-design solutions with a proxy application than with a full-fledged application or with a benchmark code because a *proxy* is an “agile” (has several implementations, which are also change-friendly) yet simple (distills the major building blocks and workflow of the original application) domain-specific self-contained application as suggested by the ExMatEx co-design Department of Energy center [4] and the Mantevo project [6].

Molecular dynamics (MD) simulations are used in various fields, such as Chemistry and Biology [3]. A MD simulation consists of a number of particles interacting with each other as an N-body system. The high number of particles in the system, and the methods used to compute forces, make MD simulations compute intensive demanding parallel implementations. CoMD is a parallel MD proxy application developed by the ExMatEx co-design center [4], and is part of the Mantevo project [6] at Sandia laboratories. The CoMD distribution includes three different versions: sequential, pure distributed-memory parallel model with MPI and hybrid distributed/shared memory parallel model with MPI/OpenMP. Both pure distributed and hybrid implementations use a domain decomposition that distributes the particles among the MPI processes. In the hybrid version, OpenMP threads support the main thread (MPI process) with computation responsibilities.

Many techniques are used to improve performance in parallel applications. Overlapping communication with computation is commonly used, and there are several ways to implement it. In [2], Becker *et al.* present three patterns to overlap communication with computation. The first one, over-decomposition, consists in fragmenting the local data in order to get chunks available for communication while the remainder is being prepared (computed). This technique may be beneficial but, in many cases, it leads to increasing the amount of communication due to fragmentation. The sec-

ond pattern presented in [2] is to use non-blocking communication. Non-blocking communication will allow proceeding to computation while messages are being received. For this to be successfully implemented, it is crucial that the following computations do not need the information that is concurrently being received. In MD applications, it is not easy to know where the data from neighboring processors are needed because particles move randomly in the domain. The third pattern is speculation, which is a method where computation does not wait until communication is done. In other words, there is no verification if it is safe to proceed with computations. For a classical MD simulation this pattern is not acceptable due to the need for high accuracy at each iteration.

It is already well known that hybrid MPI/OpenMP programming model is beneficial in multi-core environments. Not only may the hybrid model be beneficial to obtain a high degree of parallelism and to utilize the compute resources efficiently, but also it may help to reduce memory access latencies and contention if, e.g., the hybrid MPI/OpenMP execution is mapped within a compute node in accordance with the NUMA architecture of the node as was shown in the authors earlier work, e.g. [13]. Another advantage of the hybrid MPI/OpenMP model is its flexibility in letting several OpenMP threads to issue the communication calls within the same MPI process, provided a sufficient level of thread safety available in a given MPI implementation [5]. The majority of MPI implementations support four [5] levels of thread-safety, which governs how threads within an MPI process may call MPI routines at a given time. For example, the highest level, `MPI_THREAD_MULTIPLE`, allows all threads to call MPI routines with no restrictions, whereas the lowest, `MPI_THREAD_SINGLE`, is very restrictive since it allows to *execute* only one thread.

This paper proposes two algorithms: one that uses two parallel threads to manage communication and buffer handling mechanisms, respectively, and another that relies on the highest level of the MPI thread-safety support to communicate by two threads while using the rest of the available threads to unload the receive buffers.

The remainder of this paper is organized as follows. Section titled “Related Work” is a review of the work performed previously by others. Section titled “Existing Hybrid Implementation” presents the main structure of the CoMD code. Section titled “Details of the Proposed Strategies” describes the main contributions followed by the one titled “Numerical Experiments”, which presents experimental results while Section “Conclusion” summarizes and discusses future work.

RELATED WORK

Previous publications present works on communication and computation overlapping, in codes similar to CoMD using hybrid MPI/OpenMP models. In [8], Kumar *et al.* developed a communication/computation overlapping mechanism in Nano Scale Molecular Dynamics (NAMD), a MD simulation. The technique used is specific to the IBM Blue Gene/L, working around the architecture to proceed to computation for a number of processor cycle while data is transmitted

across the network. Preissl *et al.* proposed a communication/computation overlapping algorithm for a Gyrokinetic Tokamak Simulation in [12]. The code deals with particles like in MD simulations and features hybrid parallelization with MPI and OpenMP. They exploited MPI thread safety level in order to allow one OpenMP thread to communicate while the other threads concurrently work on computations. Kaiser and Baden presented a work based on a hybrid implementation of a stencil grid program in [7]. Their work partitions the domain owned by a process into two: one requiring communication and the second not requiring communication. The partition that does not require communication is computed while one of the threads is communicating. All threads have a portion of the computations they are responsible for, but one of them is given a smaller chunk in order to allow it to be freed for communication. All these works are based on hybrid implementations which is the case for CoMD. They all require a good balance on computation versus communication.

EXISTING HYBRID IMPLEMENTATION

CoMD follows the workflow of a general classical MD application: initialize particles, then, iteratively, compute velocities, positions, interaction forces for each particle, update the metrics, such as temperature and potential energy, and finally, increment the time step.

The particle domain is a 3D box within which all particles are contained and move. This domain is formed by “unit boxes” called linkcells containing the particles. Linkcells are determined by subdividing the local spatial domain with a Cartesian grid where the grid spacing in each direction is at least as big as the cutoff distance of the potential [9]. The particle-domain is split among the total number of processes according to the process-space configuration (1D, 2D, or 3D). Each process is in charge of the computation of the velocity, position, and force of its particles in the domain. All of these computations are done by a number of loops that are parallelized using OpenMP threads. After computing velocities and positions, the process has to update the position of each particle and proceed to communication according to the process-space configuration. For a 1D configuration, left (x^-) and right (x^+) boundaries (called faces) are communicated with neighbors and the remaining faces (y^- , y^+ , z^- , and z^+) are sent to self. In a 2D configuration, processes need to communicate left and right, front and back (faces x^- , x^+ , y^- , and y^+), the z^- and z^+ are sent to self. For a 3D configuration, processes communicate all six faces with their neighbors¹. Figure 1 shows the communication faces (dashed) for four-process-domain in a 1D configuration. The arrows show where the faces are sent. The update of particle locations and communication (halo exchange) are done sequentially, thus keeping all the remaining threads idle. Once halo exchange is done, the next task is to compute forces. It is important to

¹The 1D process-space is $n \times 1 \times 1$; the 2D process-space configuration is $n \times m \times 1$ and the 3D process-space configuration is $n \times m \times p$, where n , m , and p are integers representing numbers of processes: n —in the x direction, m —in the y direction, and p —in the z direction.

note that the exchange is needed not only because the particles have moved but also because each process needs a certain number of particles belonging to its neighbors (within a minimal radius for interaction also known as the cutoff distance) in order to achieve the force computations. Since in CoMD, faces have to be communicated in a specific order, the usability of threads when it comes to communication may be hindered in part due to the hierarchical nature of the communications. As shown in Fig. 2, there are portions of faces to be communicated overlapping with each other (see shaded squares in the domain corners). And the hierarchy in the CoMD communications is brought about due to these overlapping regions: Faces x^- and x^+ have to finish communicating before faces y^-, y^+, z^- , and z^+ do so, because the part of the data received for the x faces has to be incorporated into the y prior to their communication. Correspondingly, parts of the y faces are needed in the z faces before the latter may be sent. Notice that, $-$ faces may be sent concurrently with their $+$ counterparts. However, the existing implementation of CoMD does not take advantage of this possibility.

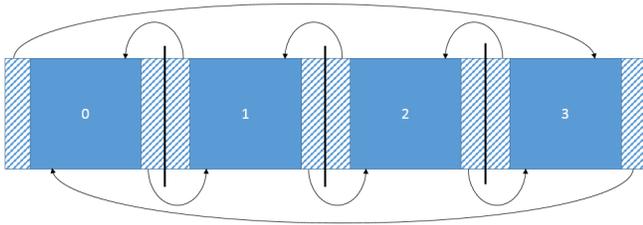


Figure 1. 1D process-space configuration.

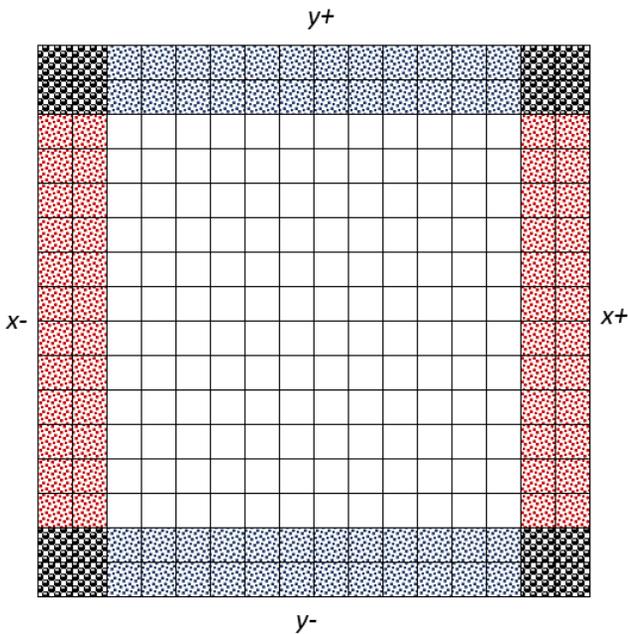


Figure 2. Halo exchange portions for faces x^-, x^+, y^- , and y^+ .

DETAILS OF THE PROPOSED STRATEGIES

This section details two different communication overlapping strategies and presents the corresponding algorithms, which exploit two concurrent threads in the halo exchange and all available threads to unload buffers.

Reorganization of particle update

After velocity and position computations, the particles have to be moved to their new positions. The mechanism used to reorganize the particles within each linkcell has opportunities for improvement. In the original implementation of CoMD (see Section “Existing Hybrid Implementation”), only a single thread is used to reorganize particles by moving them to either local or halo boxes before communication starts, which hinders the parallelism. Instead, it is possible to use all the available threads, which is especially beneficial for larger numbers of particles, i.e., for problem scaling.

To update all the boxes, the multi-threading is performed in a `parallel-for` fashion, such that each thread has a chunk of data equal to the total number of boxes divided by the number of threads. If a particle does not belong to a current box, it has to be moved out. For example, Fig. 3 shows a domain that is divided into three chunks to accommodate the three available threads. Therefore, a `parallel-for` loop would proceed as follows: thread 0 works on boxes 0 through 27, thread 1 works on boxes 28 through 55 and thread 2 will work on boxes 56 through 83. Each thread will start from the smallest box ID to the highest.

0	13	14	27	28	41	42	55	56	69	70	83
1	12	15	26	29	40	43	54	57	68	71	82
2	11	16	25	30	39	44	53	58	67	72	81
3	10	17	24	31	38	45	52	59	66	73	80
4	9	18	23	32	37	46	51	60	65	74	79
5	8	19	22	33	36	47	50	61	64	75	78
6	7	20	21	34	35	48	49	62	63	76	77

Figure 3. Particle positions update by three threads (four columns of particle boxes per thread).

In the best-case scenario, the particles move to the neighboring rather than to “remote” boxes. However, this is not always the case. Furthermore, it may occur that several threads try to copy a particle to the same box simultaneously, thus, a caution in implementation is necessary to prevent race conditions.

Halo exchange

As was mentioned earlier, an efficient way to exchange halo particles among neighboring processes is to take advantage of thread parallelism, which is absent for communication in the original version of CoMD. However, it is necessary to schedule threads bearing in mind the dependency among faces to be exchanged. Two strategies are proposed and implemented below for multi-threaded halo exchange.

Two-Task Strategy

This strategy takes advantage of the fact that two faces can be sent per axis. Since the dependency is between axis and not faces of the same axis, two threads can be used to load, communicate, and then unload for each axis while the remaining threads are idle. In this strategy only one thread, thread # 0, is in charge of all the MPI communications. In particular, thread 0 starts by loading face f^- and then sends it. Concurrently, thread 1 supports thread 0 with the second task—hence, the strategy is termed as *two-task* (TT)—of loading face f^+ . Then thread 1 waits until thread 0 hands it the received buffer for face f^+ to be unloaded after completing communication of the face f^- . Thread 0 goes on with sending face f^+ , previously loaded by thread 1 and then unloads the received buffer for face f^- . The advantage of this technique is that both threads get some work done unlike in the original version in which everything was done by thread 0 and the other threads were all idle. In most cases however, loading and unloading of the buffers would take less time than communicating the faces. Algorithm 1 implements the TT strategy. The use of `barrier` is needed to synchronize the threads in such a way that thread 0 would start communicating the buffer loaded by thread 1 only when the later has completed its job. Note that the TT strategy creates a load imbalance between the two threads since the communication is typically longer than buffer handling, resulting in a wait time for the non-communicating thread.

```
for each axis do
  if threadID=0 then
    load send_buffer for face  $f^-$ 
    send face  $f^-$  and receive face  $f^+$ 
    #barrier
    send face  $f^+$  and receive face  $f^-$ 
    unload receive_buffer for face  $f^-$ 
  else
    load send_buffer for face  $f^+$ 
    #barrier
    unload receive_buffer for face  $f^+$ 
  end
end
```

Algorithm 1: Pseudo-code for the two-task strategy.

Two-Peer Strategy

This strategy aims to correct the load imbalance that the TT strategy may exhibit. Instead of having the two threads working in the lock step, each of the two threads is completely independent and works entirely on communicating a particular face. Hence, this strategy was named *two-peer* (TP). In particular, thread 0 loads face f^- , sends it and receives face f^+ . Concurrently, thread 1 loads face f^+ sends it and receives face f^- . After threads 0 and 1 finished communicating, all the available threads jump in and work together to unload the received data from buffers. An advantage of the TP strategy is that thread 0 and 1 are more balanced compared with the TT Strategy because both have the same type of work to accomplish. Unfortunately, all the three axes cannot be processed in parallel due to sequential axis treatment in the CoMD while avoiding this axis ordering entails significant

modifications to the CoMD code. Algorithm 2 implements the TP strategy. The use of `barrier` is again needed to synchronize the threads in the end of their respective communications. Note that an MPI implementation of the TP strategy necessitates the highest level of the MPI thread support, i.e., `MPI_THREAD_MULTIPLE`, which allows all threads to call MPI routines.

```
for each axis do
  if threadID=0 then
    load send_buffer for face  $f^-$ 
    send face  $f^-$  and receive face  $f^+$ 
    #barrier
  else if threadID=1 then
    load send buffer for face  $f^+$ 
    send face  $f^+$  and receive face  $f^-$ 
    #barrier
  else
    #barrier
  end
  unload receive_buffer for face  $f^+$ 
  unload receive_buffer for face  $f^-$ 
end
```

Algorithm 2: Overlapping communication with buffer handling: Two threads communicate.

NUMERICAL EXPERIMENTS

The experiments were performed on Hopper a Cray XE6 system, resource of the National Energy Research Scientific Center (NERSC). Hopper has 6384 compute nodes equipped with two twelve-core AMD ‘MagnyCours’ 2.1-GHz processors with total memory of 32 GB per node. A processor has two dies directly connected to a quarter of the total memory on the node each. Each die has six cores. The die and the part of the memory to which it is connected constitute a NUMA node [10]. The existence of a NUMA architecture may allow a hybrid program to run efficiently, if the threads and MPI processes are mapped in accordance with the NUMA boundaries. For example, the most efficient mapping on a Hopper node is when four MPI processes are executing, one per NUMA node, and each having six threads for 24 threads total per Hopper node. Hopper nodes are connected to each other in a 3D torus mesh fashion, using a custom chip to route communication over the network known as ‘Gemini’. The network is proprietary to Cray, and often referred to as ‘Cray Gemini Network’ [11]. The Gemini network interface controller (NIC) has a maximum bandwidth of 8.3 GB/s per direction [11].

Obtaining latency and bandwidth parameters.

Bandwidth and latency tests have been conducted for the MPI implementation used in the experiments. Hopper uses a proprietary MPI implementation, Cray MPICH version 6.0.1, based on the ANL MPICH. Cray MPICH implements all four thread safety levels. A simple model for communication time $T_{\text{comm}} = T_l + T_m M$ was used to obtain the communication latency T_l and the time to transmit a byte of data T_m by using linear regression. A ‘ping-pong’ type message exchange was performed between two distinct nodes for message sizes

M varying from 56 B, corresponding to a single-particle data record, to about 30 MB, which is an upper bound for a communication by a local domain. As a result, T_l was determined to be $46.05 \mu s$ and T_m to be $0.0027 \mu s$. The obtained model may be used to estimate the time to communicate each face in CoMD for “Gemini” and compare this time to the buffer handling time in the TT and TP strategies for growing problem sizes and domain configurations. Additionally, for the TP strategy, the communication time will depend on the implementation of the `MPI_THREAD_MULTIPLE` mode.

Problem setup.

To test the weak scaling of the proposed strategies, a total of five problem-size/core-count combinations (denoted $C1, \dots, C5$), in millions of atoms and number of cores respectively, were tested: (96,144), (192, 288), (384, 576), (768, 1152), and (1600, 2400). Specifically, to obtain each combination, the number of linkcells per process was kept the same while the grid size (number of cores) grew.

Results.

The results are organized in Table 1, representing the total execution times across the implementations tested, and in Figs. 4 to 6, representing performance gains as percentage of the particle exchange in TT and TP as compared with the original CoMD hybrid implementation (denoted here as MP) for 1D, 2D, and 3D domains. Note that, in Table 1, the bold-faced values represent best times among domain configurations for each problem-size/core-count combination. A general observation is that the two proposed versions kept the scalability property of the original code (see Table 1). In particular, the execution times remain almost constant for all the implementations, which also indicates that the use of locks in TT and TP to prevent race conditions did not impact the performance of the code significantly.

The atom update times, measured separately for each combination and domain configurations, showed up to 62% performance increase in the proposed strategies as compared to the original one. Specifically, all the MP measurements were about 30 seconds each and each of the proposed strategies was measured as 11.5 seconds, which is 62% better when all the six available threads participate in the atom update as opposed to a single thread.

The TT and TP strategies produced substantial improvements over the original (MP) version in the halo exchange routines. For TT, Figs. 4 to 6 show gains between 9.09% and 61.34%. The TP strategy produced even better gains, ranging between 31.11% and 72.95%. The main reason for disparities between the gains in TT and TP is in the fact that TT creates a load imbalance between the two threads since the communication is typically longer than buffer handling. Thus, the second thread tasked with buffer handling incurs some wait time between buffer loading and unloading operations. Another reason for disparities between TT and TP is that, in TP, the unloading of buffers is shared among the available threads, working in parallel. Notice that the performance gains are lower with lower core counts, that they reach their maximum around 500 cores, and then decrease for the largest core count tested here. The decrease is especially pronounced for the 2D and 3D

domain configurations and may be attributed to the rapidly growing communication-to-computation ratio so that the effect of mutli-threading in halo exchange diminishes.

Observe the difference in time it takes to simulate 1D, 2D, or 3D problems. The 1D domains require the least amount of communication across the network, thus, spending less time to communicate. Therefore, for the same problem size, a 1D configuration will almost always be faster than 2D or 3D configurations, and a 2D configuration will almost always be faster than a 3D one. The time difference is due only to the particle exchange, everything else takes the same amount of time. An increased communication to computation ratio also influences negatively the performance gains in the halo exchange (cf. Figs. 4 to 6) because at most two threads may be used to send and receive data.

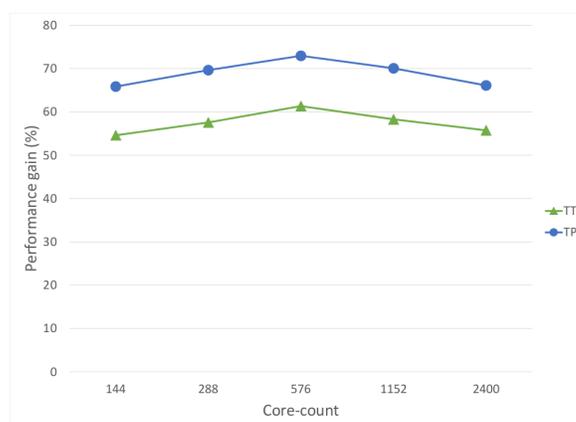


Figure 4. Performance gains over the original implementation in halo exchange for 1D configuration.

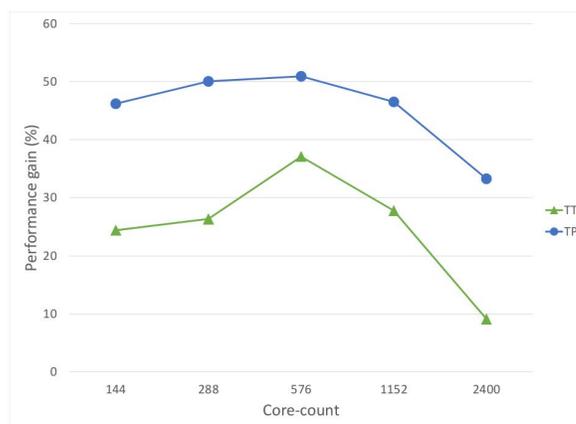


Figure 5. Performance gains over the original implementation in halo exchange for 2D configuration.

Problem/Core Combination	1D			2D			3D		
	MP	TT	TP	MP	TT	TP	MP	TT	TP
<i>C1</i>	413.12	389.55	388.76	421.51	391.39	389.75	413.86	393.37	392.05
<i>C2</i>	413.09	389.22	388.39	411.53	391.13	389.39	413.49	401.69	391.37
<i>C3</i>	414.70	389.38	388.46	413.06	391.12	390.71	416.04	393.92	392.16
<i>C4</i>	413.19	389.72	396.69	411.13	390.77	389.61	415.36	394.81	392.71
<i>C5</i>	413.74	389.55	389.12	414.49	395.62	392.9	416.87	396.19	394.36

Table 1. Execution time (in seconds) for different problem-size/core-count combinations and domain configurations.

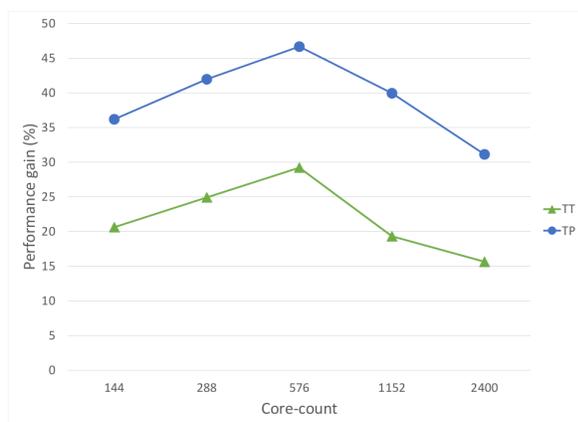


Figure 6. Performance gains over the original implementation in halo exchange for 3D configuration.

So far, the performance gains have been discussed specifically for the halo exchange. It is important to examine the impact of these improvements on the overall execution time. Recall that the force computation is the most time consuming part of an MD simulation. However, having gains of up to 72.95% in the atom exchange will have a direct impact on the overall simulation time. Here, TT produced an overall best time improvement of 25.32 seconds over MP on 576 cores, observed with the 1D configuration, which is a 6.1% improvement. TP produced an even better best-overall performance gain, which was also observed over 576 cores with the 1D configuration, of 26.24 seconds, which is a 6.32% improvement. The lowest gain observed for TT is 11.8 seconds, over 288 cores with a 3D configuration, this translates as a 2.85% improvement of simulation time. TP on the other side performed the lowest for a 1D configuration over 1152 cores with a performance gain of 16.5 seconds corresponding to a 4% improvement in simulation time. It is important to note that this lowest gain for TP is not accurate due to the fact that the overall time includes the time spent for setting up the simulation, which can be longer than expected at times.

CONCLUSION

Typical research agendas to improve MD simulations often downplay the communication overhead for particle exchanges because most MD simulations spend close to 70% their time in the force computation. In contrast, this work attempts to improve the performance of a proxy MD appli-

cation, CoMD, by exploring strategies to hide communication based on multi-core node architectures. Two specific strategies were proposed and implemented. The first strategy uses two threads concurrently for the two distinct tasks: One thread communicates and loads/unloads part of the buffers while another loads and unloads the other part of the buffers. The second takes a more balanced approach and considers the two threads as peers: both do buffer handling and communication of particles. Numerical experiments, performed on Hopper at NERSC, showed up to 73% of performance gains in the communication routines for the proposed strategies. These gains translate into about 6% of the overall execution time savings. Future work will involve extending the developed strategies developed with tunable parameters to configure CoMD and other MD applications to a variety of networks and different NUMA layouts in multi-core node architectures.

Acknowledgments

This work was supported in part by the Air Force Office of Scientific Research under the AFOSR award FA9550-12-1-0476, and by the National Science Foundation grants NSF/OCI—0941434, 0904782, 1047772. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. The authors thank the reviewers for their insightful comments and valuable suggestions to improve the paper.

REFERENCES

- Barrett, R. F., Dosanjh, S. S., Heroux, M. A., Hu, X. S., Parker, S., and Shalf, J. Towards Codesign in High Performance Computing Systems. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, San Jose, CA (2012).
- Becker, A., Venkataraman, R., and Kale, V. L. Patterns for Overlapping Communication and Computation. In *Workshop on Parallel Programming Patterns (ParaPLOT 2009)* (2009).
- ExMatEx. CoMD Proxy Application. www.exmatex.org/comd.html.
- ExMatEx. DoE Exascale Co-Design Center for Materials in Extreme Environments. www.exmatex.org.
- Gropp, W., and Thakur, R. Thread-safety in an MPI implementation: Requirements and analysis. *Parallel Computing* (2009).

6. Heroux, M. A., Doerfler, D. W., Willenbring, P. S. C. J. M., Edwards, H. C., Williams, A., Rajan, M., Keiter, E. R., Thornquist, H. K., and Numrich, R. W. Improving Performance via Mini-applications. Tech. rep., Sandia National Laboratories, 2009.
7. Kaiser, T. H., and Baden, S. B. Overlapping Communication and Computation with OpenMP and MPI. *Scientific Programming* (2001).
8. Kumar, S., Huang, C., Zheng, G., Bohm, E., Bhatele, A., Phillips, J. C., Yu, H., and Kale, L. V. Scalable Molecular Dynamics with NAMD on the IBM Blue Gene/L system. *IBM journal of research and development* (2008).
9. Mohd-Yusof, J., and McPherson, A. ExMatEx/CoMD. <https://github.com/exmatex/CoMD/blob/master/src-mpi/linkCells.c>.
10. NERSC. Compute Nodes. <http://www.nersc.gov/users/computational-systems/hopper/configuration/compute-nodes/>.
11. NERSC. INTERCONNECT. <http://www.nersc.gov/users/computational-systems/hopper/configuration/interconnect/>.
12. Preissl, R., Koniges, A., Ethier, S., Wang, W., and Wichmann, N. Overlapping Communication with Computation using OpenMP tasks on GTS Magnetic Fusion Code. *Scientific Programming* (2010).
13. Srinivasa, A., and Sosonkina, M. Nonuniform memory affinity strategy in multithreaded sparse matrix computations. In *Proceedings of the 2012 Symposium on High Performance Computing, HPC '12*, Society for Computer Simulation International (San Diego, CA, USA, 2012), 9:1–9:8.

Shared-Memory Parallelization of the Semi-Ordered Fast Iterative Method

Josef Weinbub¹
weinbub@iue.tuwien.ac.at

Florian Dang^{2,3}
fdang@aneofr

Tor Gillberg^{4,5}
torgi@simula.no

Siegfried Selberherr¹
selberherr@iue.tuwien.ac.at

ABSTRACT

The semi-ordered fast iterative method is used to compute a monotone front propagation of anisotropic nature by solving the eikonal equation. Compared to established iterative methods, such as the fast iterative method, the semi-ordered fast iterative method (SOFI) offers increased stability for variations in the front velocity. So far, the method has only been investigated in a serial, two-dimensional context; in this paper we investigate a parallel implementation of SOFI (using OpenMP) and evaluate the method for three-dimensional real-world type problems. We discuss the parallel algorithm and compare its performance and its computed solutions with an OpenMP-powered fast iterative method. Different speed functions together with varying problem sizes are used to investigate the impact of the computational load. Although the semi-ordered fast iterative method is inferior to the fast iterative method with respect to parallel efficiency, we show that its execution performance is significantly faster.

Author Keywords

Semi-ordered fast iterative method; fast iterative method; eikonal equation; front propagation, OpenMP

ACM Classification Keywords

G.1.0 NUMERICAL ANALYSIS: General—Parallel algorithms

INTRODUCTION

Simulating an expanding front is a fundamental step in many computational science and engineering applications, such as image segmentation [5], brain connectivity mapping [12], medical tomography [11], seismic wave propagation [13], geological folds [8], semiconductor process simulation [18], or computational geometry [15].

¹Institute for Microelectronics, TU Wien, Gußhausstraße 27-29/E360, 1040 Wien, Austria

²ANEOf, 122 Ave du General Leclerc, 92100 Boulogne Billancourt, France

³Laboratoire PRISM, Université de Versailles, 45 Ave des Etats-Unis, 78035 Batiment Descartes, France

⁴Bank of America Merrill Lynch, 2 King Edward St, EC1A 1HQ London, England

⁵Simula Research Laboratory, P.O. Box 134, 1325 Lysaker, Norway

In general, an expanding front originating from a start position Γ is described by its first time of arrival T to the points of a domain Ω . This problem can be described by solving the eikonal equation [14], which for n spatial dimensions reads:

$$\begin{aligned} \|\nabla T(\mathbf{x})\|_2 &= f(\mathbf{x}) & \mathbf{x} \in \Omega \subset \mathbb{R}^n \\ T(\mathbf{x}) &= g(\mathbf{x}) & \mathbf{x} \in \Gamma \subset \Omega \end{aligned}$$

T is the unknown solution (i.e. first time of arrival), f is an inverse velocity field (i.e. $f(\mathbf{x}) = 1/F(\mathbf{x})$), and g are boundary conditions for Γ . Generally speaking, isosurfaces to the solution represent the position of the front at a given time, and can thus be regarded as the geodesic distance relative to Γ . If the velocity $F = 1$, the solution $T(\mathbf{x})$ represents the minimal Euclidian distance from Γ to \mathbf{x} .

The most widely used method for solving the eikonal equation is the fast marching method [14]. This method is inherently sequential and attempts to parallelize it have been unsatisfactory [11]. Other approaches for solving the eikonal equation include the fast sweeping method (FSM) [19][20] and the fast iterative method (FIM) [10]. Both methods support parallel execution; FIM supports fine-grained parallelism, thus inherently offering greater potential for parallelism over the entire spectrum than the coarse-grained parallelism of FSM. FIM was originally implemented for parallel execution on Cartesian meshes and later extended to triangular surface meshes [6]. FIM relies on a modification of a label correction scheme coupled with an iterative procedure for the mesh point update. The inherent high degree of parallelism is due to the ability of processing all nodes of an active list (i.e. narrow band) in parallel, thus efficiently supporting a single instruction, multiple data parallel execution model. Therefore, FIM is suitable for implementations on highly parallel accelerators, such as graphics adapters [10][11]. Although FIM has been primarily investigated regarding fine-grained parallelism on accelerators, investigations on shared-memory approaches have also been conducted [3][4][18].

Although FIM provides superior parallel performance to other available methods (in most cases), its performance is problem dependent. Complex speed functions tend to increase significantly the solution time.

To overcome this shortcoming, the semi-ordered fast iterative (SOFI) method has been developed [7]; SOFI is based on both the FIM as well as on the Two-Queue method [1]. SOFI enforces an ordering to get the iterative behavior closer to front tracking methods, i.e., fast marching and wavefront tracking methods, in turn offering an increased stability, when faced with intricate speed functions. Front tracking methods inherently favor sequential execution, therefore parallel scalability is expected to be inferior to that of the FIM. Rather than computing all *active* nodes in parallel (as in FIM), the SOFI method pauses some of the awaiting updates according to a cutoff criterion based on statistical in-situ analysis of the solution values. Therefore, the computational resources are not fully used, limiting the potential for parallel speedup relative to the FIM. However, SOFI offers excellent performance for two-dimensional, sequential problems. In turn, the Two-Queue method also pauses nodes to get a partially ordered technique, but it is only applicable to isotropic problem formulations, whereas the SOFI method supports also anisotropic problems.

This work investigates the SOFI method for three dimensional problems of varying sizes and complexity based on an OpenMP parallelization. A short overview of the original SOFI method is provided, followed by a discussion of our parallel SOFI algorithm and a set of benchmarks which are used to assess the parallel execution performance of SOFI relative to a reference FIM implementation.

THE SEMI-ORDERED FAST ITERATIVE METHOD

Let X denote the set of nodes on which we want to compute the time of arrival in a solution list T , and assume that the initial distance is known at the nodes $\Gamma \subset X$. Initially, we assume that the front does not reach any nodes which are not initialised, i.e., $T(x) = \infty, \forall x \in X \setminus \Gamma$. The current front is described by the active list, aL , containing source points. In the initialisation step, all initialised nodes are added to aL . The list of nodes, containing nodes which are ahead of the source points, the paused list, pL , is initially empty but will eventually contain paused nodes, i.e., nodes that at some point will be used as source points. All nodes in aL are used as source points to evolve the front.

The solution is constructed in a semi-ordered fashion using a cutoff parameter av , which depends on the average solution m_k (k refers to the iteration counter) of the nodes in pL . Assume that node x_n receives a new solution value that is smaller than the old value, $t_{new} < T(x_n)$ and that in addition $t_{new} \leq av$ then x_n is added to the end of aL , and thus used as a source point. If instead $t_{new} > av$, we postpone its function as a source by adding x_n to pL . When there are no source points left, no nodes in X can get a lower arrival time. Figure 1 schematically depicts the evolution of the aL and pL container.

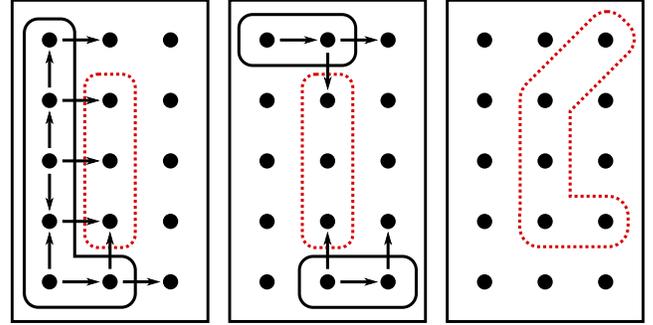


Figure 1: The evolution of the active list (black, solid boxes) and paused list (red, dashed boxes) for different iteration steps (from left to right) is shown schematically. Arrows point from nodes in aL to nodes being re-evaluated. Nodes in the active list are gradually updated until the active list is empty. The nodes in the paused list have a higher solution value than the applied cutoff value.

PARALLEL ALGORITHM

The parallel algorithm follows the original SOFI algorithm [7] to a large extent, albeit offering additional handling of shared-memory parallel programming aspects and an advanced cutoff method suitable for three-dimensional problems. We first introduce the required general data objects, discuss the initialization step and the actual parallel algorithm, and finally conclude with an analysis of the developed semi-automatic cutoff criterion.

Additionally to the originally used algorithm entities, for our parallel approach we propose a temporary aL_{temp} container to avoid expensive deletion processes of aL during the compute-intensive iterations. Also, we use a counter (aL -swaps) to track the number of swaps between aL and aL_{temp} . An essential aspect of the algorithm is the determination whether a node has been already added to aL or pL . To avoid an expensive lookup step, which would require finding the node in question within aL or pL , we use a tag-based system. To that end, we employ the aL_{tags} and pL_{tags} data structures, which provide us with element-based tag lookup for the expense of additional memory overhead. The coefficients c , $relax$, m_k , and m_{k-1} are required for our improved automatic cutoff computation, which will be explained later on.

The initialization of the parallel SOFI algorithm sets all coefficients to zero (e.g. av , aL -swaps, t_{sum}), and the solution field is preloaded with an arbitrarily high number (e.g. 10^{12}). However, the solution of each source point is initialized with zero, whereas the source nodes themselves (representing the input of the algorithm) are added to the active list aL . This is different from FIM, where neighbours of source nodes (rather than source nodes) are added to the active list.

Upon convergence of the algorithm, the result of the algorithm (i.e. the signed distance field) is stored in the solution list T .

Algorithm 1 introduces the actual parallel SOFI algorithm. The main parallel loop is - as with the FIM - processing the active list aL . We use a *guided* scheduling method, as it has shown to be the best performing scheduling procedure, due to the irregular workload inside the parallel loop demanding a dynamic load balancing. The tag system ensures that the same nodes are not added to aL/pL again during an iteration (Lines 3,9,10,19,20,42). However, nodes might be reprocessed later on during a subsequent iteration, such is the general procedure of iterative methods. The use of write guards in form of atomic locks has been minimized to three spots (Line 10,20,25). The neighbor (nb) iteration is required to generate the required 7-point stencil (Line 4), which is used to discretize the eikonal equation's differential operator in three dimensions (Line 6). Parallel write access to the pL and aL_{temp} data structures has been realized via thread-exclusive containers (Lines 13,21), which - although requiring a serial merging step at the end of the parallel for loop (Line 29,41) - scales better for increasing thread numbers than guarding central data structures with additional critical sections. A similar technique is used for the cutoff procedure's essential coefficients t_{sum} and t_{sqsum} (Lines 11,12,15,16). The thread-exclusive aL_{temp} and pL containers are merged into their global counterparts in serial *merge* steps (Lines 29,41), and are reset (i.e. the arrays are cleared) to prepare for subsequent iterations.

For the SOFI method to perform well, the algorithm for computing the cutoff coefficient av is vital, as av controls the assignment of a node to either aL or pL (Line 8,18). The cutoff level enforces an ordering of the nodes to be updated, in order to reduce the number of iterations needed. When too many nodes are activated (i.e. added to aL), the number of computations is high and the numerical solvers are slow. Similarly, if too many nodes are paused (i.e. added to pL), too few nodes are computed, as the ordering is too strict. Empirical investigations have shown best performance when approximately 80% of the nodes are activated [1][7].

The original method used for computing av is based on the average solution value of the paused nodes [7]. However, this approach does not perform well for general problems in three dimensions. The ordering enforced from this simple method tends to be too weak, since too many nodes are activated by being put into aL . When that happens, the additional cost of ordering computations outgrows its benefits. Therefore, we use a different method to compute the cutoff av , being based not only on the average solution value m_k but also on the standard deviation σ of the nodes in pL .

Algorithm 1 SOFI Parallel Algorithm

```

1: while  $aL \neq \emptyset$  do
2:   for all  $x \in aL$  #parallel, guided do
3:      $aL_{tags}(x) = 0$  #guard
4:     for all  $x_{nb}$  of  $x$  #edge-connected do
5:       if  $T(x) < T(x_{nb})$  #downwind condition
6:         then
7:            $t_{new} = SolveEikonal(x_{nb})$ 
8:           if  $t_{new} < T(x_{nb})$  then
9:             if  $t_{new} > av$  then
10:              if  $pL_{tags}(x_{nb}) == 0$  then
11:                 $pL_{tags}(x_{nb}) = 1$  #guard
12:                 $t_{sum} += t_{new}$ 
13:                 $t_{sqsum} += t_{new} \cdot t_{new}$ 
14:                Add  $x_{nb}$  to  $pL$ 
15:              else
16:                 $t_{sum} += t_{new} - T(x_{nb})$ 
17:                 $t_{sqsum} += t_{new} \cdot t_{new} - T(x_{nb}) \cdot T(x_{nb})$ 
18:              end if
19:            else
20:              if  $aL_{tags}(x_{nb}) == 0$  then
21:                 $aL_{tags}(x_{nb}) = 1$  #guard
22:                Add  $x_{nb}$  to  $aL_{temp}$ 
23:              end if
24:            end if
25:           $T(x_{nb}) = Min(T(x_{nb}), t_{new})$  #guard
26:        end if
27:      end for
28:    end for
29:    Merge ( $aL_{temp}$ ); Swap ( $aL, aL_{temp}$ ); Reset
30:    ( $aL_{temp}$ )
31:    if  $aL$ -swaps  $> \sqrt[n]{Size(X)}/10$  then
32:       $av = 0.0$ 
33:    end if
34:    if  $av > m_k$  then
35:      if  $pL$ -ratio  $< 0.5\%$  and  $aL$ -swaps  $> 5$  then
36:         $c = 0.8c$ ;  $av = m_k - relax$ 
37:      else if  $pL$ -ratio  $> 99\%$  and  $aL$ -swaps  $< 5$  then
38:         $c = 2.0c$ ;  $av = m_k - relax$ 
39:      end if
40:    end if
41:    if  $aL == \emptyset$  then
42:      Merge ( $pL$ ); Swap ( $aL, pL$ );
43:      Swap ( $aL_{tags}, pL_{tags}$ ); Reset ( $pL$ )
44:       $m_k = \sum t_{sum} / Size(aL)$ 
45:       $\sigma = \sqrt{\sum t_{sqsum} / Size(aL)}$ 
46:       $relax = c(2(m_k - m_{k-1}) + \sigma)^2 / 6\sigma^2$ 
47:       $av = m_k + relax$ ;  $m_{k-1} = m_k$ 
48:       $t_{sum} = t_{sqsum} = 0.0$ 
49:    end if
50:  end while

```

Assuming a normal distribution of the solution values of nodes in pL , we would activate approximately 84% by assigning a cutoff av of the average plus a standard deviation. However, a large spread (i.e. large σ) within pL indicates that a stricter ordering is needed. We estimate the average shift in cutoff level, by the difference between the current m_k and the previous: $\Delta m_k = m_k - m_{k-1}$. The original SOFI relaxation method is to have a cutoff level as a *relaxed* average by using $av = m_k + 1.5\Delta m_k$. Trying to factor in the indications from σ we have found that the following cutoff performs well (Lines 43-46):

$$av = m_k + c \frac{(2\Delta m_k + \sigma)^2}{6\sigma^2}$$

The additional coefficient c and *relax* are used as additional parameters to adjust the cutoff computation, by investigating the pL -ratio, i.e., the number of nodes added to pL relative to the total number of nodes added to pL and aL (Lines 33-39). The used thresholds and coefficients have been shown to work best for the presented examples, but may be adjusted for more realistic devices, hence the designation *semi-automatic*.

Another mechanism to ensure that the computed cutoff level is reasonable is proposed, which is based on monitoring the number of iterations (Lines 30-32). If too many iterations are detected, it is assumed that the cutoff level is not optimal. Therefore, the cutoff procedure is restarted by triggering a recomputation of the cutoff level, increasing the chance of upholding a high convergence rate.

BENCHMARKS

We investigate the performance of our parallel SOFI implementation relative to a reference FIM implementation. Our benchmarks cover different three-dimensional problems with varying problem sizes (100^3 and 200^3 Cartesian cube grids), speed functions, and single/multiple-source configurations (a single center source node versus 100 source nodes spread over the entire simulation domain).

Regarding speed functions, we investigate three different configurations: (1) constant speed (F_{const}), where for the entire domain $F = 1$ is used; (2) checkerboard speed (F_{check}), where the computational domain is divided into eleven equally sized cubes in each direction and the velocity is alternated between $F = 1$ to $F = 2$ from cube to cube [2][8]; (3) oscillatory speed (F_{osc}), where the speed function is modeled by a highly oscillatory continuous speed function, being $F = 1 + \frac{1}{2}\sin(20\pi x)\sin(20\pi y)\sin(20\pi z)$ [2]. The benchmark platform is a dual-socket node with two Intel Xeon E5-2620 (SandyBridge EP) 6-core (2 threads per core) processors with 128 GB of main memory, powered by a 64-bit GNU/Linux and the GNU/GCC compiler 4.9.2. The parallel algorithm introduced in the previous section has been implemented in C++. The presented execution performances are based on the median of five execution timings. The threads have been pinned to the

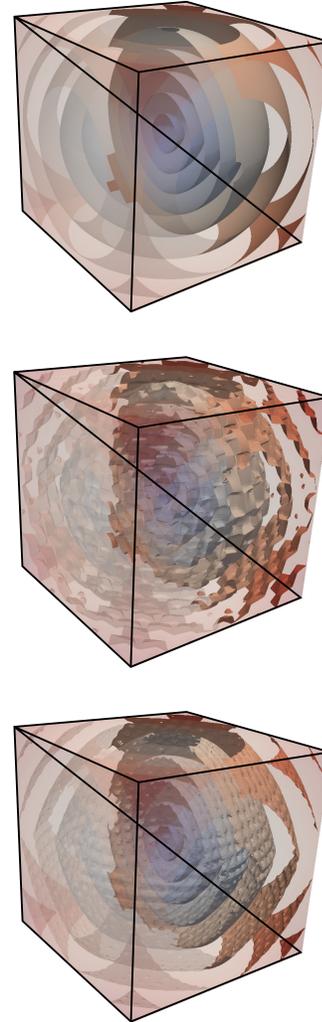


Figure 2: Isosurfaces of the F_{const} (top), F_{check} (middle), and F_{osc} (bottom) solution on a 100^3 domain for a single center source

individual physical cores via the likwid [17][16] library to avoid thread-core reassignments, which would otherwise potentially introduce a performance penalty.

Figure 2 depicts the isosurfaces of the solutions of the center test configurations for the 100^3 simulation grids, to provide a frame of reference for the benchmark setup and the solutions. The results for the 200^3 are similar, albeit offering an increased resolution. To verify the correctness of the solutions, the FIM and the SOFI method results of the single source problem with constant speed for a 100^3 grid have been compared to an analytic solution given by the Euclidian distance function. The error norms of both methods are the same, being $L_1 = 29 \cdot 10^{-3}$, $L_2 = 10^{-3}$, and $L_\infty = 36 \cdot 10^{-3}$, indicating that the SOFI method computes the same result as the FIM. If indeed the results would be different, the ε used in the FIM's algorithm can prevent full convergence of the algorithm.

Figures 3-5 compare the execution times and the parallel efficiency between our SOFI method and FIM implementation for a 100^3 grid. A logarithmic scaling is used to highlight the variances in the execution time, especially relevant for high thread numbers. The SOFI method outperforms the FIM both for the single and multiple source configurations; for the more important multiple source setups (as these cases resemble real-world applications more closely) and 12 threads, a speed-up factor of 1.5 for F_{const} , 2 for F_{check} , and 1.7 for F_{osc} is achieved. The parallel efficiency of the single source test setups is by far inferior to the FIM, although both methods suffer in general from efficiency limiting factors typical for stencil computations, being cache misses and memory latency [9]. This stems from the fact that the SOFI method inherently does not favor single source problems, as in this case no ordering is needed, thus introducing unnecessary overhead. However, for the more important multiple source cases, the scalability is reasonable: for 12 threads efficiencies of around 60% can be achieved for the highly challenging F_{check} and F_{osc} problems. The results show load balancing problems, which can be identified by the somewhat erratic parallel efficiency behavior. This fact is to be attributed to an unbalanced utilization of the aL and pL containers, triggered by insufficiencies in our automatic cutoff calculation.

Figures 6-8 continue the investigation for an increased computational domain size, being 200^3 , which allows to judge the performance under increased load. Again, execution timings show that the SOFI method is faster than FIM. For the multiple-source cases and 12 threads, a speed-up factor of 1.9 for F_{const} , 2 for F_{check} , and 2.6 for F_{osc} is achieved. The parallel efficiency is comparable

to the 100^3 grid results, being around 60% for the F_{check} and F_{osc} problems.

Overall, the previously mentioned inferior parallel potential of the SOFI method relative to the FIM is reflected in the results, albeit being still reasonable, especially for more relevant multiple source scenarios. However, the execution time is what matters in real world applications. In this light, the parallel SOFI method is significantly superior to the parallel FIM, underlining the potential of the parallel SOFI method as a compelling alternative for solving the eikonal equation in applied numerical simulations arising from the diverse field of computational science and engineering.

CONCLUSION

An approach for parallelizing the SOFI method via a shared-memory OpenMP technique has been introduced. An alternative cutoff method supporting three-dimensional problems has been discussed, for semi-automatically driving the applied iterative Two-Queue technique. Our parallel SOFI algorithm offers superior execution performance relative to a reference FIM implementation for different speed functions and problem sizes, while offering reasonable parallel efficiency. This work shows the excellent capabilities of the SOFI method for tracking front propagation paving the way for further investigations regarding real-world applications.

ACKNOWLEDGMENTS

This work has been supported by the Austrian Science Fund (FWF) through the grant P23296. The authors thank Joachim Schöberl for providing access to a dual-socket Intel Xeon system for benchmarking purposes.

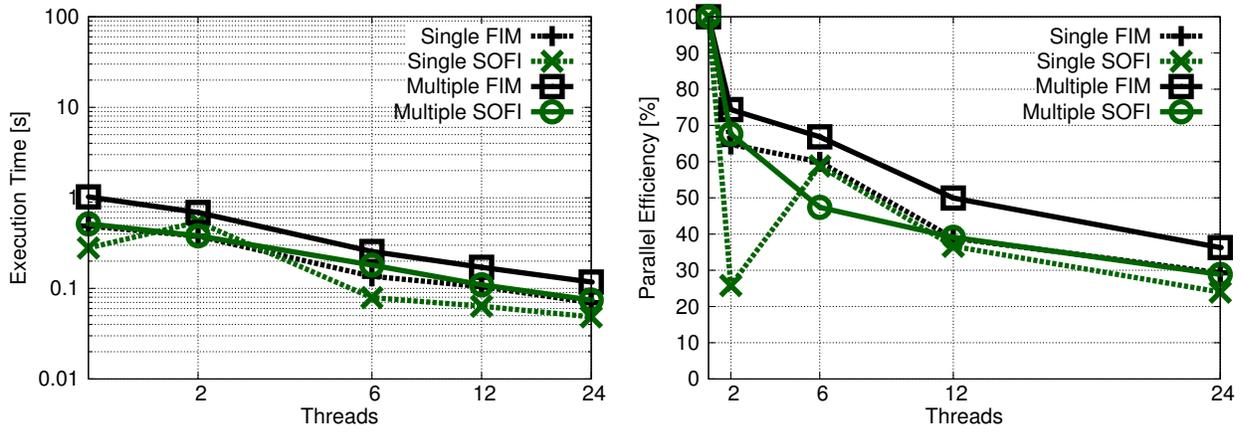


Figure 3: Execution times (left) and parallel efficiencies (right) of the F_{const} problem on a 100^3 domain

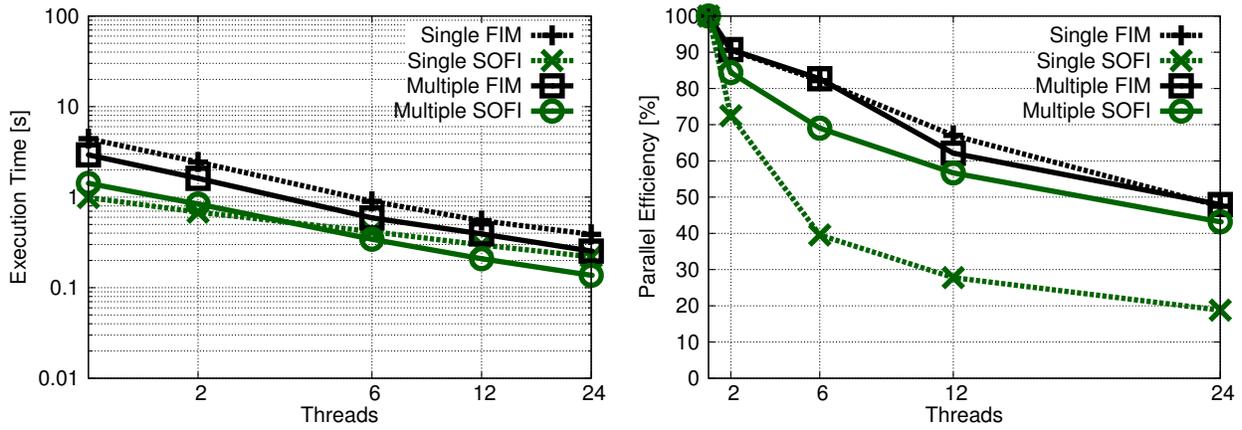


Figure 4: Execution times (left) and parallel efficiencies (right) of the F_{check} problem on a 100^3 domain

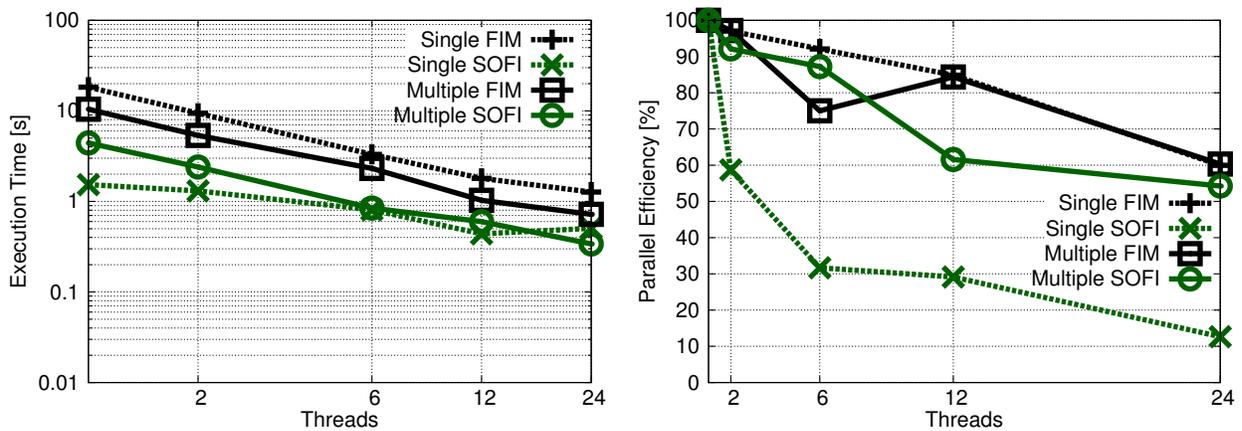


Figure 5: Execution times (left) and parallel efficiencies (right) of the F_{osc} problem on a 100^3 domain

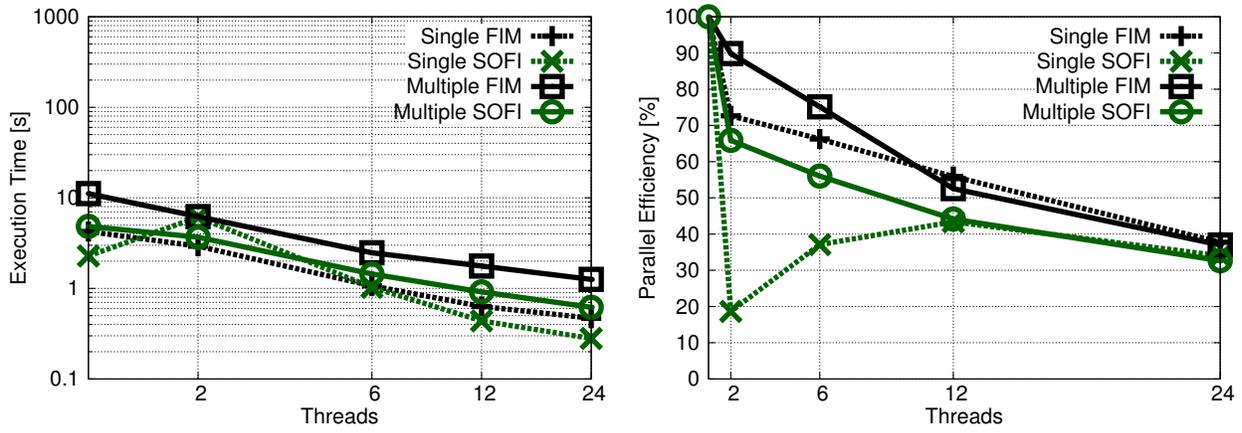


Figure 6: Execution times (left) and parallel efficiencies (right) of the F_{const} problem on a 200^3 domain

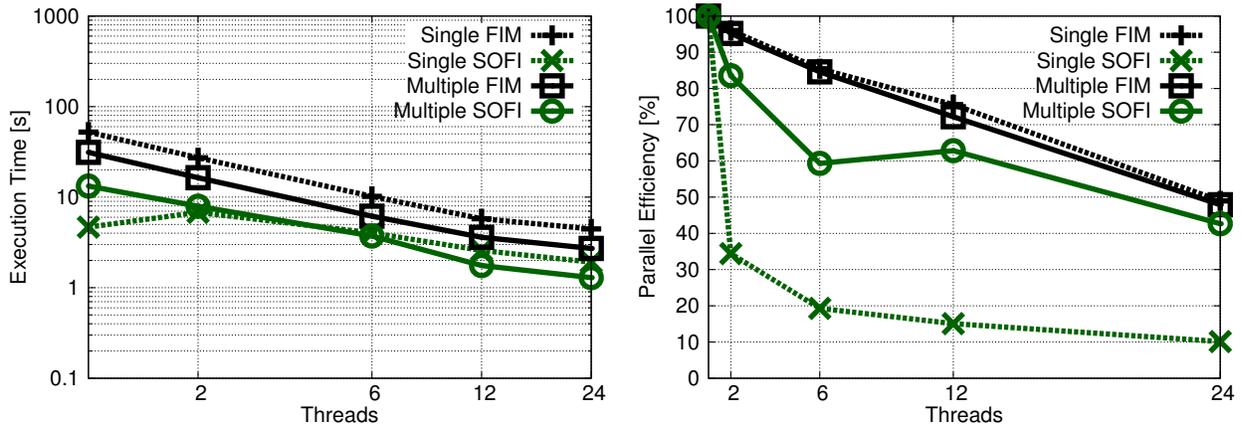


Figure 7: Execution times (left) and parallel efficiencies (right) of the F_{check} problem on a 200^3 domain

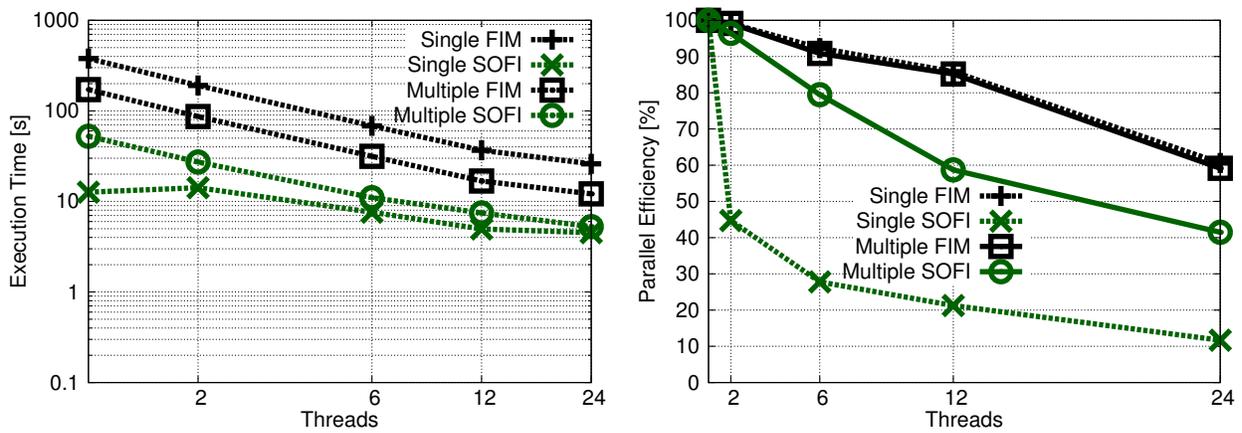


Figure 8: Execution times (left) and parallel efficiencies (right) of the F_{osc} problem on a 200^3 domain

REFERENCES

1. Bak, S., McLaughlin, J., and Renzi, D. Some Improvements for the Fast Sweeping Method. *SIAM Journal on Scientific Computing* 32, 5 (2010), 2853–2874. DOI: 10.1137/090749645.
2. Chacon, A., and Vladimirsky, A. Fast Two-Scale Methods for Eikonal Equations. *SIAM Journal on Scientific Computing* 34, 2 (2012), A547–A578. DOI: 10.1137/10080909X.
3. Dang, F., and Emad, N. Fast Iterative Method in Solving Eikonal Equations: A Multi-level Parallel Approach. *Procedia Computer Science* 29 (2014), 1859–1869. DOI: 10.1016/j.procs.2014.05.170.
4. Dang, F., Emad, N., and Fender, A. A Fine-Grained Parallel Model for the Fast Iterative Method in Solving Eikonal Equations. In *Proceedings of the International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)* (2013), 152–157. DOI: 10.1109/3PGCIC.2013.29.
5. Forcadel, N., Le Guyader, C., and Gout, C. Generalized Fast Marching Method: Applications to Image Segmentation. *Numerical Algorithms* 48, 1-3 (2008), 189–211. DOI: 10.1007/s11075-008-9183-x.
6. Fu, Z., Jeong, W. K., Pan, Y., Kirby, R., and Whitaker, R. T. A Fast Iterative Method for Solving the Eikonal Equation on Triangulated Surfaces. *SIAM Journal on Scientific Computing* 33, 5 (2011), 2468–2488. DOI: 10.1137/100788951.
7. Gillberg, T. A Semi-Ordered Fast Iterative Method (SOFI) for Monotone Front Propagation in Simulations of Geological Folding. In *Proceedings of the International Congress on Modelling and Simulation (MODSIM)* (2011), 631–647.
8. Gillberg, T., Bruaset, A. M., Hjelle, Ø., and Sourouri, M. Parallel Solutions of Static Hamilton-Jacobi Equations for Simulations of Geological Folds. *Journal of Mathematics in Industry* 4, 10 (2014), 1–31. DOI: 10.1186/2190-5983-4-10.
9. Hager, G., and Wellein, G. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, 2010. ISBN: 978-1439811924.
10. Jeong, W. K., and Whitaker, R. T. A Fast Iterative Method for Eikonal Equations. *SIAM Journal on Scientific Computing* 30, 5 (2008), 2512–2534. DOI: 10.1137/060670298.
11. Li, S., Mueller, K., Jackowski, M., Dione, D., and Staib, L. Physical-Space Refraction-Corrected Transmission Ultrasound Computed Tomography Made Computationally Practical. In *Lecture Notes in Computer Science*, vol. 5242. 2008, 280–288. DOI: 10.1007/978-3-540-85990-1_34.
12. Prados, E., Soatto, S., Lenglet, C., Pons, J.-P., Wotawa, N., Deriche, R., and Faugeras, O. Control Theory and Fast Marching Techniques for Brain Connectivity Mapping. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, vol. 1 (2006), 1076–1083. DOI: 10.1109/CVPR.2006.89.
13. Rawlinson, N., and Sambridge, M. Multiple Reflection and Transmission Phases in Complex Layered Media Using a Multistage Fast Marching Method. *Geophysics* 69, 5 (2004), 1338–1350. DOI: 10.1190/1.1801950.
14. Sethian, J. A. A Fast Marching Level Set Method for Monotonically Advancing Fronts. *Proceedings of the National Academy of Sciences* 93, 4 (1996), 1591–1595.
15. Sethian, J. A. *Level Set Methods and Fast Marching Methods*. Cambridge University Press, 1999. ISBN: 978-0521645577.
16. Treibig, J., et al. LIKWID - Webpage, 2014. <https://code.google.com/p/likwid/>.
17. Treibig, J., Hager, G., and Wellein, G. LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments. In *Proceedings of the International Conference on Parallel Processing Workshops (ICPPW)* (2010), 207–216. DOI: 10.1109/ICPPW.2010.38.
18. Weinbub, J., and Hössinger, A. Accelerated Redistancing for Level Set-Based Process Simulations with the Fast Iterative Method. *Journal of Computational Electronics* 13, 4 (2014), 877–884. DOI: 10.1007/s10825-014-0604-x.
19. Zhao, H. A Fast Sweeping Method for Eikonal Equations. *Mathematics of Computation* 74, 250 (2005), 603–627. DOI: 10.1090/S0025-5718-04-01678-3.
20. Zhao, H. Parallel Implementations of the Fast Sweeping Method. *Journal of Computational Mathematics* 25, 4 (2007), 421–429.

PerDome: A Performance Model for Heterogeneous Computing Systems

Li Tang

Department of Computer
Science and Engineering
University of Notre Dame
ltang@nd.edu

X. Sharon Hu

Department of Computer
Science and Engineering
University of Notre Dame
shu@nd.edu

Richard F. Barrett

Center for Computing
Research
Sandia National Laboratories*
rfbarre@sandia.gov

Author Keywords

Heterogeneous computing; GPU; roofline model;
performance modeling; workload partitioning;

ABSTRACT

Heterogeneous systems, consisting of different types of processors, have the potential to offer higher performance at lower energy cost than homogeneous systems. However, it is rather challenging to actually achieve the high execution efficiency promised by such a system due to the larger design space and the lack of reliable performance/energy models for aiding design space exploration. This paper fills this gap by proposing a performance model for heterogeneous systems. In processor level, the roofline model [1] can produce the performance upper bound of executed code using its ratio of computation to memory traffic. Our model, referred to as PerDome, builds on the roofline model and can reliably predict the system performance for both homogeneous execution (where each processor either executes the entire application code or none) and heterogeneous execution (where each processor executes part of the application code). Two case studies are carried out to demonstrate the effectiveness of PerDome. The results show that PerDome can indeed provide a good estimate for performance comparisons which can then be used for heterogeneous system design space exploration.

INTRODUCTION

Heterogeneous systems, consisting of different types of processors, are becoming more prevalent in not only embedded computing area (e.g., Apple iPhone 6) but also high performance computing community (e.g., Oak Ridge's Titan supercomputer). Such systems have the potential to offer higher performance at lower energy cost than homogeneous systems. However, it is rather challenging for a given application to actually achieve the high execution efficiency promised by such a system. A main difficulty of efficiently using heterogeneous systems is workload partitioning, i.e., which part of an application should run on which processor. If workload

partitioning can properly map different parts of an application to distinct processors of heterogeneous systems based on their characteristics, much higher performance can be achieved [2, 3]. Yet, most existing work on workload partitioning for heterogeneous systems (e.g. [2, 3, 4]) either considers partitioning in application level or assumes application partitioning is given in code level. Since for many applications it is not readily clear which partition can achieve the highest performance, design guidelines provided by a reliable performance model is indispensable.

A reliable performance model should be able to faithfully capture the relative performance levels of different workload partitioning options. We call a specific workload partitioning implementation a workload partition (WP). For homogeneous systems, the roofline model introduced in [1] is such a model and can be used to help design applications and homogeneous systems. However, straightforwardly employing the roofline model for heterogeneous systems, e.g., by simply aggregating two distinct processors' roofline plots, can only model some very specific execution scenarios and conditions.

In this paper, we propose PerDome to model both homogeneous and heterogeneous execution scenarios of heterogeneous systems. Similar to the roofline model, the PerDome model is based on the given processors' peak computation and memory performance, and operational intensity¹ values. We focus on two types of processors, CPU and GPU. The output of PerDome is performance upper bounds for different WPs on a given CPU+GPU heterogeneous system.

To validate our PerDome model, we conducted two application case studies. One is based on a synthetic application composed of a segment of compute-intensive code and a segment of memory-intensive code. The other is the data assembly stage (referred to as DA) in a representative FEM code (i.e., miniFE [5]). We implemented different WPs and measured their actual performance on different heterogeneous platforms. We then compared the actual and predicted relative performance levels of WPs for model validation.

Contributions.

The ultimate goal of this work is to help application developers partition workload on CPU+GPU heteroge-

*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.

¹Operational intensity is defined as the ratio of arithmetic operations per byte of data moved from/to memory [1].

neous systems for better performance. There are two main contributions in this work.

The first contribution is the consideration of a large design space. We consider, in PerDome, two processor types and a wide variety of execution models including input data-set based partitioning, code partitioning, with and without data dependencies between the processors, etc. Particularly, we propose a novel way to represent workload partitioning options on heterogeneous systems for considering all necessary information in a simple format.

Second, we use both synthetic and representative applications to validate our model. The representative application could make our work closer to real applications. The results show that PerDome can indeed provide a good estimate for performance comparisons which can then be used to aid workload partitioning design space exploration.

BACKGROUND

In this section, we first classify different execution models of using the CPU and GPU in heterogeneous systems. We then briefly review the roofline model and discuss why simple extensions of the roofline model are insufficient for predicting the performance upper bounds of WPs for the purpose of design space exploration .

Execution Models

Given a heterogeneous system and an application kernel², there are different ways (i.e., execution models) to execute the kernel. In this paper, we focus on heterogeneous systems consisting of a CPU and a GPU and assume that the application kernel has multiple input data sets to operate on. A critical design consideration is how to partition the workload between the CPU and GPU. Workload here is defined as the numbers of the floating point operations to be executed (referred to as flops) and the bytes to be moved from/to main memory (referred to as bytes). Therefore, a processor’s workload is dependent on both mapped kernels and input data sets. We classify the execution models of heterogeneous systems into four categories and illustrate some examples in Figure 1.

CPU-only and GPU-only are a simple way to use the processors of heterogeneous systems, where only one processor is responsible for the whole workload and the other processor is unused. These WPs can be desirable for achieving low-energy execution if one processor has much higher energy efficiency than the other [6]. In the data partitioning (DP³) execution model, the workload is partitioned by dividing the input data sets into two groups to be handled by the CPU and GPU separately. The granularity of the basic dividable data set unit can be

²Since an application may adopt different algorithms for different implementations, the application can have very different performance upper bounds for different implementations. Thus we use “kernel” to indicate that a specific algorithm choice for the application.

³Some papers refer to input data partitioning as workload partitioning.

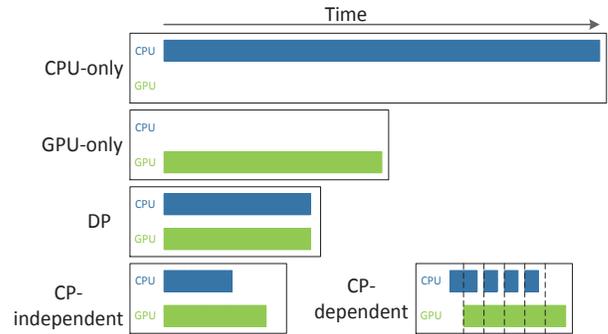


Figure 1. Four execution models of workload on heterogeneous systems. CP-independent represents the conditions that CPU and GPU execute their mapped kernel code without data communication between each other.

small and independent. Thus, perfect DP (p-DP) is assumed to be able to always finely partition the workload and distribute them evenly to CPU and GPU for perfectly overlapping the execution time of both processors. For extreme DP cases where one processor’s input data sets are empty, DP is equivalent to CPU-only or GPU-only. Some recent work (e.g., [7, 8]) studied statically partitioning of input data sets or dynamically distributing tasks for balancing CPU/GPU workload.

Though DP can balance CPU/GPU workload, it cannot fully exploit the distinct capabilities of the CPU and GPU. Another execution model is code partitioning (CP) where the CPU and GPU execute different segments of the task, i.e., parts of the kernel code. Usually the more compute-intensive part of the kernel code is offloaded to the GPU for higher execution efficiency. Depending on whether CP causes CPU/GPU data dependency, we further classify CP to CP-dependent and CP-independent (see the last two execution patterns in Figure 1). The CPU and GPU of CP-independent execute their mapped kernel code concurrently without interruption for data communication. For CP-dependent, the input data sets are partitioned into small groups and the CPU and GPU communicate after processing each group of data sets instead of all the data sets. The CPU/GPU stages are scheduled by using a simple software pipelining scheme. When the number of data set groups is large enough, CP-dependent can overlap most of the execution time of CPU and GPU. However, CP-dependent becomes much more complicated and incurs high synchronization overhead when there are more than two CPU/GPU stages. Thus we only focus on two CPU/GPU stages in our performance model.

Roofline Model

Williams et al. introduced the roofline model in [1] to derive the performance upper bound for a given processor based on the processor’s peak computation performance, memory bandwidth, and the operational intensity of any kernel code. The performance bottleneck of a kernel code can be easily obtained by using the kernel’s operational

intensity to find the corresponding performance upper bound in the roofline plot.

Let I represent the operational intensity, and is defined by

$$I = \frac{N_f}{N_b}, \quad (1)$$

where N_f and N_b are the number of floating point operations (flops) and the number of bytes moving to/from the main memory (bytes), respectively. The assumption behind the roofline model is that processors can perfectly overlap the computation and memory traffic. Hence, the processor execution time is modeled as

$$\begin{aligned} T &= \max\{N_f t_f, N_b t_b\} \\ &= N_b t_b \max\{I \frac{t_f}{t_b}, 1\}, \end{aligned} \quad (2)$$

where t_f and t_b are the average time per flop and byte movement. The performance upper bound, P , using flops/sec as the unit is calculated by

$$\begin{aligned} P &= \frac{N_f}{N_b t_b \max\{I \frac{t_f}{t_b}, 1\}} \\ &= \frac{1}{t_b \max\{\frac{t_f}{t_b}, I^{-1}\}}. \end{aligned} \quad (3)$$

Since t_f and t_b can be treated as constants for a particular processor, Equation (3) indicates that the processor is memory bounded if $I \frac{t_b}{t_f}$ is smaller than I . The roofline plot depicts this performance trend in a 2D space and uses x- and y-axes for kernel's operation intensity and performance upper bound, respectively.

The roofline model can be readily extended to predict performance upper bounds for homogeneous systems. That is, the system performance upper bound is simply the aggregation of all processors' performance upper bounds since each processor is identical and the best WP is evenly distributing data sets to all processors. The roofline model can also be easily applied to *some* execution models in heterogeneous systems. For CPU-only and GPU-only, the roofline model can be straightforwardly used to predict the system performance upper bound. In p-DP, one can add the performance upper bounds of CPU and GPU to get the system performance upper bound since CPU and GPU concurrently run the whole kernel from beginning to end.

However, for CP, it is not immediately clear how to apply the roofline model since CP can have many different WPs and their CPU and GPU might have different intensities and execution times. One possibility is to aggregate the performance upper bounds of CPU and GPU with their corresponding intensities to get the system performance upper bound. But this method can lead to upper bounds that are too loose since it does not take into consideration that CPU and GPU may finish at different times. For example, consider a CP based WP having its CPU's execution time much longer than

its GPU's. By using separate CPU and GPU intensities and the roofline model, the performance upper bounds of CPU and GPU, P_C and P_G , can be obtained. Assuming that $P_C \ll P_G$, simply aggregating P_C and P_G would result in a system upper bound much higher than the actual system performance. A tighter bound for this WP should be close to P_C since the GPU is not used for most of the system execution time. Having upper bounds that are too loose often causes misleading performance comparison results.

Overall, a good performance model for heterogeneous systems should be able to handle all the execution models and result in performance upper bounds that are tight and have high fidelity. By high fidelity, we mean that the relative magnitude of the performance upper bounds should mostly reflect the relative magnitude of the corresponding actual performance values. To our best knowledge, no such models exist for heterogeneous systems.

PERDOME MODEL

In this section, we first define three essential WP parameters, which can uniquely represent a WP on heterogeneous systems for the purpose of performance prediction. We then present the details of the PerDome model based on these parameters.

Essential Workload Partition Parameters

The essential WP parameters are the input to the PerDome model for generating performance upper bounds. There are two key considerations for selecting appropriate parameters as the essential WP parameters. First, the parameters should cover the whole workload partitioning design space and each WP should have a unique representation by using the parameters. Second, the number of such parameters should be small so that the resulting model is easy to use.

For a given heterogeneous system, an application kernel, and any given WP, according to the workload definition, $\{N_f, N_b\}$, $\{N_{f-C}, N_{b-C}\}$ and $\{N_{f-G}, N_{b-G}\}$ uniquely represent the workload of the entire application, and the workloads (numbers of flops and bytes) of the CPU and GPU, respectively. The actual values of these workload parameters are compiler and architecture dependent, but can be estimated at the algorithm level as discussed in [1]. Timing parameters (time per flop and byte), $\{t_{f-C}, t_{b-C}\}$ and $\{t_{f-G}, t_{b-G}\}$, reflect the peak computation and memory performance of the CPU and GPU, respectively, and can be obtained by using processors' specification data and running memory benchmarks.

For heterogeneous systems, the execution time is dominated by the slower processor. Based on the parameters introduced above and the concept used in Equations (2-3), the upper bound on the system performance can be calculated as

$$P = \frac{N_f}{T} \quad (4)$$

where

$$T = \max\{\max\{N_{f-C}t_{f-C}, N_{b-C}t_{b-C}\}, \max\{N_{f-G}t_{f-G}, N_{b-G}t_{b-G}\}\}. \quad (5)$$

Since the timing parameters are constant for a given hardware platform, to estimate performance upper bound, a straightforward way is to use all 6 WP parameters, i.e., N_f , N_b , N_{f-C} , N_{b-C} , N_{f-G} and N_{b-G} . However, as discussed in [1], it is more general to use operational intensities instead of raw flops and bytes.

The roofline model [1] uses a single kernel's operational intensity as its input to model performance of workload on one processor. Thus, a straightforward extension for the inputs of the PerDome model is using both the operational intensity for the CPU and that of the GPU. This choice does allow the PerDome model to cover the whole workload partitioning design space. However, it does not take into consideration the operational intensity of the entire application kernel. Given two different application kernels that have different operational intensities, it is possible for them to have WPs with exactly the same CPU and GPU intensities. If only CPU and GPU intensities are used to predict performance upper bound, the two application kernels would have the same upper bound. Now suppose one application kernel executes most of its code on the CPU while the other application kernel executes most of its code on the GPU. It is highly likely that the two application kernels have significantly different performance upper bounds. Thus, simply using the operational intensities of the CPU and GPU is not sufficient.

Let $\{I_C, I_G\}$, defined in Equations (6-7) below, be the operational intensities of the kernels assigned to CPU and GPU, respectively.

$$I_C = \frac{N_{f-C}}{N_{b-C}} \quad (6)$$

$$I_G = \frac{N_{f-G}}{N_{b-G}} \quad (7)$$

We choose to use I , I_C and I_G as the essential WP parameters for the PerDome model, where I is the operational intensity of the whole application kernel. Note, the whole application kernel is composed of the kernels assigned to CPU and GPU, and the operational intensity is zero when the corresponding processor is unused. Clearly, each WP can be represented by a particular set of the essential WP parameters.

Below, we present two observations which are useful for comparing performance upperbounds of different WPs.

LEMMA 1. *For an application kernel having intensity I , if a WP satisfies $I_C = I$ (resp., $I_G = I$), then either $I_G = I$ (resp., $I_C = I$) or $I_G = 0$ (resp., $I_C = I$).*

Proof: Note that the $I_G = 0$ and $I_C = 0$ cases correspond to the CPU-only and GPU-only WP, respectively. Hence the lemma holds. For the case of $I = I_C = I_G$, by definition, we have

$$N_f = N_{f-C} + N_{f-G}, \quad (8)$$

and

$$N_b = N_{b-C} + N_{b-G} \quad (9)$$

By combining Equations (6-7) and using the condition that $N_{f-C}/N_{b-C} = N_f/N_b$ we get

$$N_{f-C} \times N_b = N_{f-C} \times N_{b-C} + N_{f-G} \times N_{b-C} \quad (10)$$

With some manipulations, we get $N_{f-G}/N_{b-G} = N_f/N_b$. ■

LEMMA 2. *For an application kernel having intensity I , if a WP satisfies $I_C > I$ (resp., $I_G > I$), then $I_G < I$ (resp., $I_C > I$).*

This lemma can be proved in a similar fashion but with more mathematical manipulation. We omit the details due to space limit. Lemma 2 indicates that a kernel cannot be partitioned into two smaller kernels with their operational intensities both higher or lower than the original kernel's operational intensity.

Performance Upper Bound

We now derive the performance upper bound expression using the essential WP parameters. Note the performance upper bound defined in Equation (4) can be rewritten as

$$P = \max\left\{\frac{N_{f-C}}{N_f}t_{f-C}, \frac{N_{b-C}}{N_f}t_{b-C}, \frac{N_{f-G}}{N_f}t_{f-G}, \frac{N_{b-G}}{N_f}t_{b-G}\right\}^{-1} \quad (11)$$

Our goal is to express N_{f-C} , N_{b-C} , N_{f-G} and N_{b-G} as functions of the essential WP parameters, $\{I, I_C, I_G\}$. We consider two cases: (i) $I \neq I_C \neq I_G$ and (ii) $I = I_C = I_G$. According to Lemma 1 and 2, these two cases cover all the possible WPs for a given application kernel.

Consider first $I \neq I_C \neq I_G$. According to the definition of operational intensity in Equation (1), we have

$$I * N_b = I_C * N_{b-C} + I_G * N_{b-G}. \quad (12)$$

The total number of moved bytes, N_b , is equal to the sum of N_{b-C} and N_{b-G} . Thus, we can extend Equation (12) to

$$I * (N_{b-C} + N_{b-G}) = I_C * N_{b-C} + I_G * N_{b-G}. \quad (13)$$

By incorporating the essential WP parameters, $\{I, I_C, I_G\}$, we can solve Equation (13) to get the ratio of N_{b-C} to N_{b-G}

$$\frac{N_{b-C}}{N_{b-G}} = \frac{I_G - I}{I - I_C}. \quad (14)$$

By replacing N_{b-C} by $N_b - N_{b-G}$ in Equation (14), we have

$$N_{b-G} = N_b \frac{I - I_C}{I_G - I_C}. \quad (15)$$

Then N_{b-C} can be obtained by solving $N_b - N_{b-G}$

$$N_{b-C} = N_b \frac{I - I_G}{I_C - I_G}. \quad (16)$$

Similarly, we can get the numbers of flops for CPU and GPU

$$N_{f-G} = I_G N_b \frac{I - I_C}{I_G - I_C}. \quad (17)$$

$$N_{f-C} = I_C N_b \frac{I - I_G}{I_C - I_G}. \quad (18)$$

By using Equations (15-18) and the performance upper bound expression in Equation (11), we have

$$P = \max\left\{t_{f-C} \frac{I_C(I - I_G)}{I(I_C - I_G)}, t_{b-C} \frac{I - I_G}{I(I_C - I_G)}, t_{f-G} \frac{I_G(I - I_C)}{I(I_G - I_C)}, t_{b-G} \frac{I - I_C}{I(I_G - I_C)}\right\}^{-1}. \quad (19)$$

Based on the performance upper bound expression in Equation (19), we construct our PerDome model. For ease of visualization, each PerDome plot corresponds to a specific application kernel intensity I value. This choice allows the user of the model to easily compare how different WPs (I_C and I_G) impact the system performance. A PerDome plot thus forms a surface in the 3-D space where the x- and y-axes depict the operational intensities of the kernels assigned to CPU and GPU, and the z-axis depicts the corresponding performance upper bound. Consequently, for a given application kernel's operational intensity, the PerDome model readily reveals which combination of the CPU and GPU intensities leads to the highest performance upper bound.

Now consider $I = I_C = I_G$. In this case, we have $N_{f-C} = I \times N_{b-C}$ and $N_{f-G} = I \times N_{b-G}$. Using Equation (9), we can re-write the performance upper bound as

$$P = \max\left\{\left(\frac{N_{f-C}}{N_f} t_{f-C}, \frac{N_{f-C}}{N_f \times I} t_{b-C}, \left(1 - \frac{N_{f-C}}{N_f}\right) t_{f-G}, \left(1 - \frac{N_{f-C}}{N_f \times I}\right) t_{b-G}\right)\right\}^{-1} \quad (20)$$

For a given application kernel intensity I value, the performance upper bound in this case depends on the ratio of N_{f-C} to N_f . In general, the ratio is chosen such that the highest performance upper bound is achieved. Therefore, in this case there is only a single point for the PerDome plot with $I_C = I_G$.

Note that Equation (20) can also be used to obtain the performance upper bound for the cases of $I_G = 0$ and $I_C = 0$ (i.e., the CPU-only and GPU-only WPs) by simply setting N_{f-C} to N_f or 0, respectively. Thus, the performance upper bounds for these two cases correspond to two points in the PerDome plot with $I_G = 0$ and $I_C = 0$, respectively.

EXPERIMENTAL VALIDATION

In this section, we present our effort in validating the PerDome model. We first briefly discuss the case studies. We then give details on the hardware platforms used and their associated timing parameters. Lastly, we apply the

PerDome model to the two case studies and compare the trends of different WPs' actual performance and their performance upper bounds for model validation.

Case Studies

We used two case studies in our validation effort: one is a synthetic application kernel and the other is the DA kernel in miniFE [5]. A number of WPs based on different execution models are considered for each application kernel.

Synthetic Application Kernel

Since a synthetic application (SA) kernel allows one to easily control the numbers of flops and bytes in the whole application kernel as well as in the kernels assigned to CPU and GPU, we use it to help explore different WP options. To show the benefits of different WPs, we construct the SA kernel with a piece of compute-intensive code and a piece of memory-intensive code. The main component of the SA kernel is shown below.

```
#define SIZE 2560000

float a[8][SIZE], b[SIZE], c[SIZE], d[SIZE], e[SIZE];

for (int i=0; i<SIZE; i++) {
    for (int j=0; j<8; j++) b[i] += pow(a[j][i], 16); \
        PowAdd: 128 flops & 8 bytes
    e[i] = c[i] + d[i]; \
        VecAdd: 1 flop & 12 bytes
}
```

The input to the SA kernel are vectors **a**, **c** and **d**. The output of the SA kernel are vectors **b** and **e**. To stress the CPU and GPU usage, we set SIZE to be 2560000. The main loop iteratively runs two functions: PowAdd and VecAdd. PowAdd is to aggregate the sixteenth power of data items in **a** vector and has the operational intensity of 2. VecAdd is a regular vector addition of two vectors and has the operational intensity of 0.083.

We implemented four WPs based on the classified execution models. For the CPU-only WP, the implementation employs the OpenMP static scheduling to evenly distribute the main loop iterations to available threads. For the GPU-only WP, each iteration is assigned to one GPU thread. The WP of p-DP divides the main loop iterations into two parts. One runs the CPU-only executable while the other runs the GPU-only executable. The partitioning is done such that the CPU and GPU complete the execution at the the same time. Since the PowAdd and VecAdd functions in the SA kernel are data independent, we implemented the CP-independent WP, where all main loop iterations of PowAdd are run on GPU and VecAdd are on CPU.

DA in miniFE

To demonstrate how the PerDome model can help in application development, we selected the DA stage in miniFE as our second case study. As discussed in [5], miniFE is sufficiently complex and is a proxy FEM application that could be used to predict the performance trends of real-world FEM applications. The main role of DA is to generate an equation system including a large

sparse global matrix stored in the compressed sparse row (CSR) format. The original implementation of DA in miniFE has two separate functions: (i) stiffness matrix computation (STC) and (ii) stiffness matrix assembly (SMA). More details of DA in miniFE can be found in [8].

To implement the CPU-only and GPU-only WPs of DA, we follow the ideas in [8] and select **Omp** and **SmemDA** as the implementations of DA on CPU and GPU, respectively. **Omp** uses the OpenMP static scheduling to distribute the workload onto each thread evenly at the element level. In **SmemDA**, the work of one element is assigned to eight GPU threads. For the p-DP WP, we partition the workload at the element level following the performance ratio of **Omp** to **SmemDA** and concurrently run **Omp** and **SmemDA** with their corresponding elements on CPU and GPU, respectively.

Given the more complex structure of DA, different CP based WPs can be derived but it is not always clear which of these WPs would have higher performance. For our case study, we implemented two CP-dependent WPs. The first one, referred to as CP1, only offloads STC onto the GPU. The rationale behind this WP is that SMA in DA is mainly composed of irregular memory accesses and STC is mainly responsible for matrix computation. Another CP-dependent WP, referred to as CP2, tries to further move some GPU unfriendly code out of STC and offload it onto the GPU. STC has some operations, e.g., division and branching operations, that are not efficient for GPU execution. In CP2, we only offload the key computation part of STC onto the GPU. Since both CP1 and CP2 are CP-dependent, we employ the simple software pipelining scheme to schedule the CPU and GPU stages in CP1 and CP2.

Model Parameters

We use two CPUs and two GPUs to form four heterogeneous systems. The hardware details are summarized in Table 1. The problem sizes of the SA and DA kernels are 2560000 and 100^3 , respectively. The data type is single precision floating point.

We use the processors' architecture information, such as the number of cores and core base frequencies (the feature of multiple issue is not used in real execution), to compute peak computation performance. The values of t_{f-C} or t_{f-G} are set to the inverse of the corresponding processors' peak computation performance and are shown in Table 1. For the memory timing parameters, the STREAM [9] and SHOC [10] benchmarks are used to measure the peak memory bandwidth of our CPUs and GPUs. The peak memory bandwidth is used to get t_{b-C} and t_{b-G} which are summarized in Table 1.

We compute the essential WP parameters of the WPs for the SA and DA kernels and summarize them in Table 2. Here, columns 2, 3 and 4 show the operational intensities of the application kernel, kernels assigned to CPU and GPU, respectively.

Model Validation

Table 1. Processor data

Processor	# of cores	Base core clock (GHz)	Time per flop (ps)	Time per byte (ps)
i7 2600K	4	3.4	73.5	65.9
i3 2100T	2	2.5	200	73
GTX Titan	2688	837	0.4	4.1
GTX 750	512	1020	1.9	14.8

Table 2. Essential WP parameters

Workload partition	I	I_C	I_G
CPU-only (DA)	4.4	4.4	0
GPU-only (DA)	4.4	0	4.4
p-DP (DA)	4.4	4.4	4.4
CP1 (DA)	4.4	0.4	5.4
CP2 (DA)	4.4	1.5	9.2
CPU-only (SA)	1.7	1.7	0
GPU-only (SA)	1.7	0	1.7
p-DP (SA)	1.7	1.7	1.7
CP (SA)	1.7	0.1	2.0

We first use Equation (19) to calculate the performance upper bounds for all combinations of I_C and I_G values for SA and DA on our selected heterogeneous systems, i.e., i7+GTX750, i7+Titan, i3+GTX750 and i3+Titan. The timing and essential WP parameters needed by Equation (19) are given in Tables 1 and 2. Figures 2(a)-(d) and Figure 3(a)-(d) depict the performance upper bounds for SA and DA running on the four heterogeneous platforms, respectively. In all the figures, the x-, y- and z-axes represent I_C and I_G , and performance upper bounds, respectively. Both the x- and y-axes use logarithmic scale with base of 2. The graphs in these figures share some common features. Specifically, there are two separate surfaces representing performance upper bounds of CP based WPs. These WPs are in two distinct regions of combinations of I_C and I_G , which are shaded in grey in the x-y plane. The other two regions are blank since their I_C and I_G combinations have the I_C and I_G values both larger or smaller than the original application's operational intensity (which would never exist according to Lemma 2). The two surfaces have high performance upper bounds when I_G approaches 1.7 and 4.4 for SA and DA, respectively. Each figure also includes a point corresponding to p-DP running on the corresponding hardware platform. The performance upper bounds of p-DP is the aggregation of the ones of CPU-only and GPU-only. The points of CPU-only and GPU-only are omitted since p-DP has higher performance upper bounds than CPU-only and GPU-only.

We next study the tightness of the generated performance upper bounds. Based on the PerDome plots, performance upper bounds can be obtained by checking the corresponding combination of I_C and I_G values. The measured actual performance (as bars) with the corresponding predicted performance upper bounds (as markers) of SA and DA are shown in Figures 2(e) and 3(e), respectively. It can be seen that, for SA, the actual performance values of p-DP and CP are close to the pre-

dicted performance upper bounds. For example, the actual performance of CP on i3+Titan achieves 97% of the predicted performance upper bound. On the other hand, for DA, the actual performance values of WPs are significantly lower than the corresponding predicted performance upper bounds due to DA's complicated computation and memory behaviors. CP1 on i3+GTX750 has the tightest predicted performance upper bound among the others and its actual performance only achieves 43% of the predicted performance upper bound. Thus, our PerDome can generate tight performance upper bounds for applications with simple computation and memory behaviors that can fully exploit the processor power, but only provides relatively loose bounds for other cases.

To examine the fidelity of the predicted performance upper bounds, we compare the actual relative performance levels with the predicted ones for different WPs. As Figure 2(e) shows, for SA, the actual performance trends completely agree with the predicted performance upper bound trends. For example, CP has 14% and 17% higher predicted performance upper bounds and 3% and 9% higher actual performance than p-DP on i7+GTX750 and i3+GTX750, respectively. Also, the performance upper bounds of CP are 61% and 65% lower than p-DP, and the actual performance values of CP are 60% and 62% lower than p-DP on i7+Titan and i3+Titan, respectively.

Figure 3(e) also confirms the high fidelity of our PerDome model for DA. Specially, on i7+GTX750, CP1 has 7% higher predicted performance upper bound and 6% higher actual performance than p-DP. On all the other systems, the actual and predicted relative performance levels agree with that p-DP is faster than CP1 and CP1 is faster than CP2. Thus, the predicted performance upper bounds can reliably reflect the relative magnitude of the corresponding actual performance.

Using the PerDome plots, one can get some useful design guidelines. Take Figure 2(a) as an example. This figure is for executing SA on i7+GTX750. One can observe that the point corresponding to the p-DP WP has a performance upper bound (128 GFLOPS) that is lower than the top part of the right surface, which indicates that the CP based WPs in this region may have higher actual performance than p-DP. Furthermore, we can see that CP has higher actual performance upper bound (136 GFLOPS) than p-DP.

SUMMARY

We have proposed a performance model, PerDome, for heterogeneous systems by extending the roofline model introduced in [1]. PerDome is applicable to a variety of execution models and uses a minimal set of WP parameters. To validate the effectiveness of our proposed model, we carried out two detailed case studies. Specially, we implemented a number of different WPs for the SA and DA kernels on four heterogeneous systems, and measured the actual performance of these implementations. Comparisons between the actual performance and the predicted performance upper bounds show that the

PerDome model can generate tight performance upper bounds and provide reliable relative performance levels for heterogeneous systems.

REFERENCES

1. S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.
2. W. Sodsong, J. Hong, S. Chung, Y. Lim, S.-D. Kim, and B. Burgstaller, "Dynamic partitioning-based jpeg decompression on heterogeneous multicore architectures," in *PMAM*. ACM, 2014, p. 80.
3. M. Daga and M. Nutter, "Exploiting coarse-grained parallelism in b+ tree searches on an apu," in *SCC*. IEEE, 2012, pp. 240–247.
4. K. Ma, X. Li, W. Chen, C. Zhang, and X. Wang, "Greengpu: A holistic approach to energy efficiency in gpu-cpu heterogeneous architectures," in *ICPP-41*. IEEE, 2012, pp. 48–57.
5. R. Barrett, P. Crozier, D. Doerfler, M. Heroux, P. Lin, H. Thornquist, T. Trucano, and C. Vaughan, "Assessing the Validity of the Role of Mini-Applications in Predicting Key Performance Characteristics of Scientific and Engineering Applications," *Journal of Parallel and Distributed Computing*, vol. 75, 2014.
6. L. Tang, X. S. Hu, and R. F. Barrett, "Performance and energy implications for heterogeneous computing systems: A minife case study," *Sandia National Laboratories, Tech. Rep. SAND2014-20215*, 2014.
7. C.-K. Luk, S. Hong, and H. Kim, "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *MICRO-42*. IEEE, 2009, pp. 45–55.
8. L. Tang, X. S. Hu, D. Z. Chen, M. Niemier, R. F. Barrett, S. D. Hammond, and G. Hsieh, "Gpu acceleration of data assembly in finite element methods and its energy implications," in *ASAP-24*. IEEE, 2013, pp. 321–328.
9. J. D. McCalpin, "Stream: Sustainable memory bandwidth in high performance computers," Tech. Rep., 1991-2007.
10. A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (shoc) benchmark suite," in *GPGPU-3*. ACM, 2010, pp. 63–74.

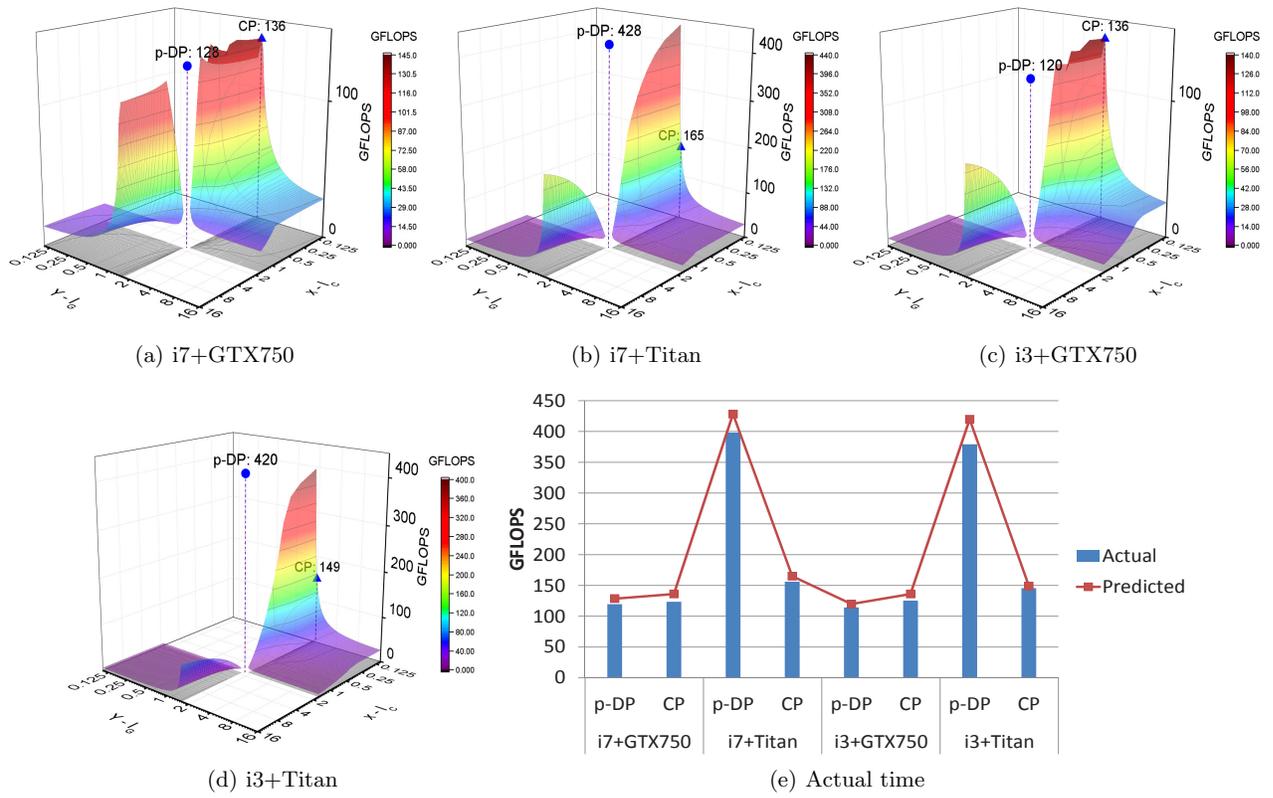


Figure 2. Performance upper bounds and actual execution time of executing SA on four different platforms.

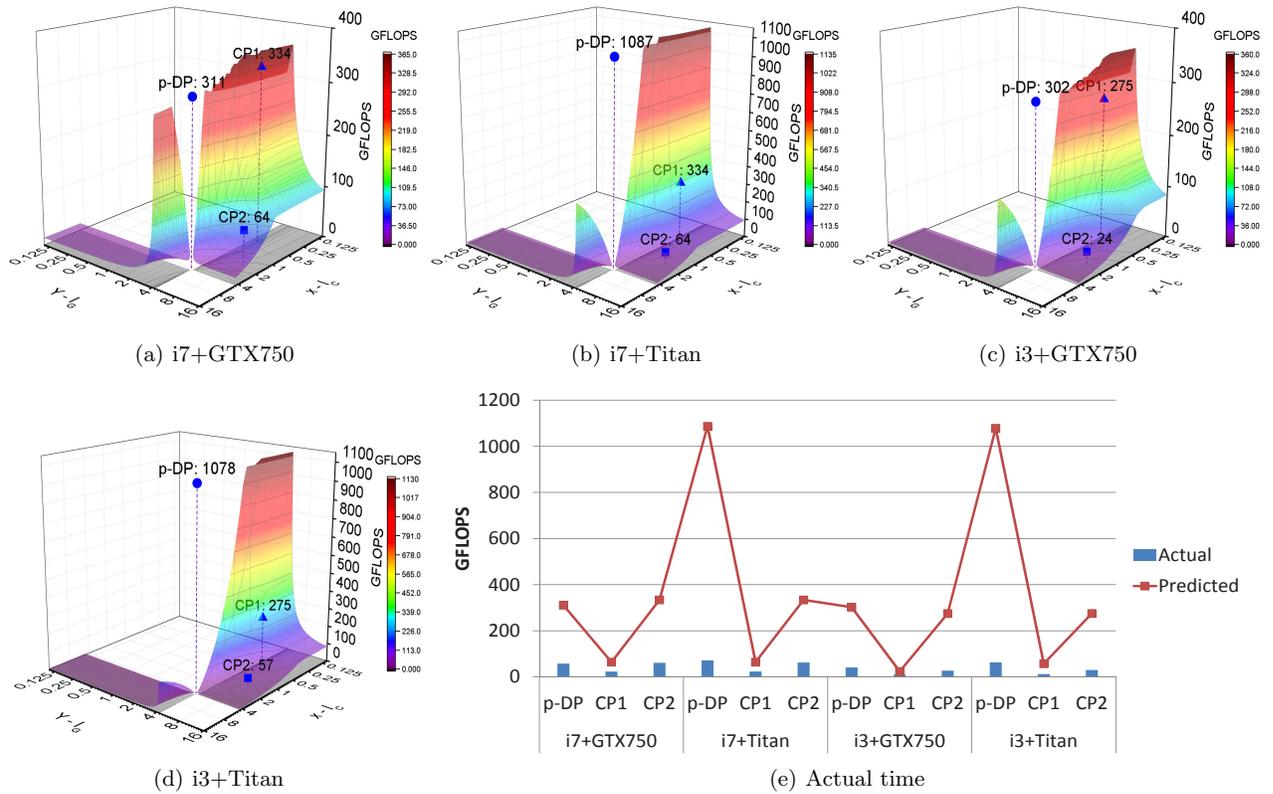


Figure 3. Performance upper bounds and actual execution time of executing DA on four different platforms.

An Improved Probability-One Homotopy Map for Tracking Constrained Clustering Solutions

David R. Easterling

Department of Computer
Science
Virginia Polytechnic Institute
and State University
Blacksburg, VA 24061
drest@vt.edu

Layne T. Watson

Departments of Computer
Science, Mathematics, and
Aerospace and Ocean
Engineering
Virginia Polytechnic Institute
and State University
Blacksburg, VA 24061

N. Ramakrishnan

Department of Computer
Science
Virginia Polytechnic Institute
and State University
Blacksburg, VA 24061

ABSTRACT

This paper proposes a new homotopy map for use with constrained clustering problems, improving over previously introduced maps through the introduction of a K-Means approximation and the use of the Kreisselmeier-Steinhaus function to provide more efficient computation, as well as demonstrating through experimentation the power of this new map.

Author Keywords

probability-one homotopy, constrained clustering, machine learning, semisupervised learning

ACM Classification Keywords

I.5.3 CLUSTERING: Algorithms

INTRODUCTION

As machine learning permeates multiple fields of science and engineering, new objective functions are continually being proposed to suit the demands of new application domains. Multicriteria objective functions especially are becoming more prevalent in areas such as mixing labeled and unlabeled data [15], incorporating constraints [20], and transfer learning [21]. The difficulty of solving such problems, and even interpreting the solution in a meaningful way, is likewise a growing field of research. A mechanism for solving problems in one field may or may not be adaptable to solving problems in another, and the information yielded by the method may not contain everything needed by the modern researcher.

One such multiobjective formulation is in the area of constrained clustering. In constrained clustering [2], the goal is not just to obtain clusters that are local in their respective spaces but that also obey a discrete set of a priori must-link

(ML) and cannot-link or must-not-link (MNL) constraints between points. More complicated constraint sets can be represented in simpler form by these ML and MNL constraints, as well; in particular, the conventional cluster hypothesis may be enforced through the use of ϵ - and δ - constraints [7]. Although there are many powerful constrained clustering algorithms published in the literature, there is currently a lack of a systematic mathematical theory to guide the design of formulations and understand the tradeoffs that invariably result as each algorithm attempts to serve two masters.

The fundamental problem in algorithm design for constrained clustering problems is the tradeoff between conventional clustering objectives and the requirements of the linking constraints. Broadly speaking, there have been two types of algorithms designed to deal with this problem [9]. The first uses the constraints to learn a distance function. The second strictly enforces the constraints as the algorithm iterates to a useful solution. The primary problem motivating the development of these two algorithmic approaches is that determining the feasibility of a set of constraints that contains both MNL and ML constraints is an NP-complete problem, being equivalent to the graph coloring problem. When the existence of a feasible solution can not be determined in polynomial time, the usual approach is to fall back on heuristics, with the hope that the resulting solution will be good enough. This dovetails nicely with one of the dominant viewpoints of big-data ML, where rigorous solutions are usually impractical due to the sheer amount of data involved, NP-complete or not, and as such heuristic approaches are the norm for reaching reasonable solutions in a reasonable time.

One traditional solution to such heuristically solved biobjective problems is to introduce a parameter λ that balances or weights competing considerations, in this case cluster locality versus constraint satisfaction. Although there are interesting theoretical insights into the complexity of constrained clustering problems [6], there is little existing theory available that can deal with (1) how to efficiently compute solutions parametrically as λ varies, (2) how to find and deal with multiple solutions for a fixed λ , and (3) how to canonically define the best choice of λ . Furthermore, using such λ as an independent variable often poses insurmountable problems to the researcher, as will be shown.

Homotopy methods are systematic approaches to characterize solution sets by smoothly tracking solutions from one formulation to another (in this case, from an unconstrained formulation to a constrained formulation). This can allow the effect of changing λ on the quality and nature of the solutions to be mathematically characterized. Smoothly tracking solutions as λ varies provides a holistic understanding of the interplay between the algorithm and a dataset. The resulting tradeoff curve can yield information about the nature of the problem and the probability of improvement offered by constraints.

Corduneanu and Jaakkola [5] used classical continuation to study how two diverse information sources should be combined in order to arrive at an integrated model. The first application of modern homotopy methods to machine learning was by [11], who showed that a general semisupervised formulation for hidden Markov models (HMMs) can be realized using a probability-one homotopy as well. [10] developed the first probability-one homotopy map for constrained clustering solutions. That paper also presented a look at the parallel performance of this algorithm with relation to other algorithms used for constrained clustering. The key contribution here is to develop that map further, introducing a K-Means approximation and the Kreisselmeier-Steinhauser function to approximate multiple inequality constraints. Finally, some experimental results are presented to demonstrate the viability of this map.

BACKGROUND

Let superscripts denote vector indices and subscripts denote components of vectors and scalar indices unless otherwise indicated. Let all norms be 2-norms unless otherwise indicated and let all distances be Euclidean distances. Let \mathbb{R}^n denote n -dimensional real Euclidean space and let $\mathbb{R}^{n \times m}$ be the set of real $n \times m$ matrices. Let the i th row of a matrix $A \in \mathbb{R}^{n \times m}$ be denoted by A_i and the j th column by $A_{.j}$. Finally, for a vector $x \in \mathbb{R}^n$, $x > 0$ means all $x_i > 0$, $x \geq 0$ means all $x_i \geq 0$, and $x \geq 0$ means $x \geq 0$ but $x \neq 0$.

Given a set $\hat{X} = \{x^i \mid x^i \in \mathbb{R}^d, i = 1, 2, \dots, k\}$ of k points (cluster representatives) in d dimensions, let $\bar{X} = \text{vec}(x^1, x^2, \dots, x^k) \in \mathbb{R}^{kd}$. Given a set $\hat{Y} = \{y^i \mid y^i \in \mathbb{R}^d, i = 1, 2, \dots, n\}$ of n data points in d dimensions, let $Y = \text{vec}(y^1, y^2, \dots, y^n) \in \mathbb{R}^{nd}$. Represent a constraint by the vector $c = (a, b, z, w) \in \mathbb{R}^{2d+2}$ of two data points $a, b \in \hat{Y}$, an identifier $z = \pm 1$, and a degree-of-belief weight $\mathbb{R} \ni w > 0$, where an identifier of $z = 1$ means that a and b are bound by a must-link constraint (i.e., must be in the same cluster) and an identifier of $z = -1$ means that a and b are bound by a cannot-link constraint (can not be in the same cluster). Given a set $\hat{C} = \{c^i \mid c^i \in \mathbb{R}^{2d+2}, i = 1, 2, \dots, q\}$ of q constraints, let $C = \text{vec}(c^1, c^2, \dots, c^q) \in \mathbb{R}^{q(2d+2)}$.

Penalty Function and Constraints

For a data point $y \in \hat{Y}$ and two cluster prototypes $x^i, x^j \in \hat{X}$ define the comparator function $D : \mathbb{R}^d \times \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R}$ by

$$D(x^i, x^j, y) = (\max\{0, \|x^i - y\|^2 - \|x^j - y\|^2\})^4.$$

Note that D is three times continuously differentiable, $D \geq 0$, and $D(x^i, x^j, y) > 0$ if and only if the distance between y and x^i is larger than the distance between y and x^j .

Given $a, b \in \hat{Y}$, let the must-link function $F_m : \mathbb{R}^d \times \mathbb{R}^d \times \mathbb{R}^{kd} \rightarrow \mathbb{R}$ be defined by $F_m(a, b, X) =$

$$\prod_{i=1}^k \left(\sum_{j=1, j \neq i}^k D(x^i, x^j, a) + D(x^i, x^j, b) \right)$$

and let the cannot-link function $F_c : \mathbb{R}^d \times \mathbb{R}^d \times \mathbb{R}^{kd} \rightarrow \mathbb{R}$ be defined by $F_c(a, b, X) =$

$$\sum_{i=1}^k \left(\prod_{j=1, j \neq i}^k D(x^j, x^i, a) D(x^j, x^i, b) \right).$$

Then the following observations are easily verified.

Observation 1. F_m and F_c are nonnegative and three times continuously differentiable.

Observation 2. For any must-link constraint $c = (a, b, 1, w) \in \hat{C}$, the must-link function $F_m(a, b, X) = 0$ if and only if constraint c is satisfied.

Observation 3. For any cannot-link constraint $c = (a, b, -1, w) \in \hat{C}$, the cannot-link function $F_c(a, b, X) = 0$ if and only if constraint c is satisfied.

Observation 4. The penalty function

$$F(C, X) = \sum_{\{i: z_i=1\}} w_i F_m(a^i, b^i, X) + \sum_{\{i: z_i=-1\}} w_i F_c(a^i, b^i, X)$$

is zero if and only if all the constraints in \hat{C} are satisfied.

By Observation 4, if it is possible to satisfy all of the constraints, then there exists a vector of cluster representatives \mathcal{X} such that $F(C, \mathcal{X}) = 0$. This vector of cluster representatives represents a global minimum point of the function F at which $\nabla_X F(C, \mathcal{X}) = 0$. Unfortunately, if multiple $x^i \in \hat{X}$ coalesce, the resulting D values will result in zero even though there is no clear clustering arising from this case, and $\min_X F(C, X)$ has the trivial solutions $x^1 = \dots = x^k$. Thus it is necessary to constrain the optimization problem $\min_X F(C, X)$ to prevent such a degenerate case from occurring. In addition, X should be bounded, as $\lim_{\|X\| \rightarrow \infty} F(C, X) = 0$ is possible.

First, consider the bounding constraint. A straightforward concave function $\Psi : \mathbb{R}^{kd} \rightarrow \mathbb{R}$ to achieve bounding is $\Psi(X) = B - \sum_{i=1}^n \|x^i\|^2 \geq 0$, where $B \in \mathbb{R}$ is a given large constant. Second, to prevent the degenerate condition noted above, a set of constraints $g_i : \mathbb{R}^{kd} \rightarrow \mathbb{R}$ can be constructed as $g_i(X) = \epsilon_g - \|x^{i1} - x^{i2}\|^2 \leq 0$, where $1 \leq i \leq \binom{k}{2}$,

$x^{i_1}, x^{i_2} \in \hat{X}$ are different cluster representatives and $\epsilon_g > 0$ is a small constant. Note that these constraints are differentiable everywhere, and satisfy the reverse convex constraint qualification at \mathcal{X} if $\Psi(\mathcal{X}) > 0$ is inactive. If the active constraints at \mathcal{X} satisfy a constraint qualification (e.g., Arrow-Hurwicz-Uzawa), then the resulting optimization problem

$$\begin{aligned} & \min_X F(C, X) \\ & \text{subject to } -\Psi(X) \leq 0, \\ & g_i(X) \leq 0, \quad 1 \leq i \leq \binom{k}{2} \end{aligned} \quad (1)$$

satisfies the Karush-Kuhn-Tucker (KKT) necessary conditions at \mathcal{X} .

Now that the problem has been defined, it remains to show how homotopy methods can be used to effect a smooth transition from a traditional clustering to a clustering that solves the above optimization problem, yielding a useful tradeoff curve. First, however, several tools necessary to utilize the homotopy method are described.

Kreisselmeier-Steinhausser Function

Since there are $\binom{k}{2}$ separation constraints in the optimization problem, an aggregation function that can reduce these to a single inequality constraint is of benefit. The Kreisselmeier-Steinhausser envelope function [12]

$$Z(X) = \frac{1}{\kappa} \ln \left[\sum_{i=1}^{\binom{k}{2}} \exp(-\kappa g_i(X)) \right],$$

where $\kappa > 0$ is a regularization parameter, is a common aggregation function used in optimization to reduce the number of inequality constraints to one. Let $g_{\max}(X) = \max_{1 \leq i \leq \binom{k}{2}} g_i(X)$. Note that

$$g_{\max}(X) \leq Z(X) \leq g_{\max}(X) + \frac{\ln(k(k-1)/2)}{\kappa},$$

which means that if $Z(X) \leq 0$ then $g_i(X) \leq 0$ for all i . As an approximation function, however, there are some drawbacks. The selection of κ is important; a large κ will result in a small difference between the value of Z and g_{\max} , but may also cause some numerical difficulties. Furthermore, the feasible region defined by Z is generally smaller than the feasible region defined by g_{\max} , as should be obvious from the above inequalities. While each g_i is concave, Z is neither quasiconcave nor pseudoconvex. Nevertheless, except for the degenerate case

$$\nabla \Psi(\mathcal{X})(\nabla Z(\mathcal{X}))^T = \|\nabla \Psi(\mathcal{X})\| \|\nabla Z(\mathcal{X})\|$$

the constraints $-\Psi(X) \leq 0$, $Z(X) \leq 0$ satisfy the Arrow-Hurwicz-Uzawa constraint qualification at \mathcal{X} . A KKT point \bar{X} for

$$\min_X F(C, X)$$

$$\begin{aligned} & \text{subject to } -\Psi(X) \leq 0, \\ & Z(X) \leq 0 \end{aligned} \quad (2)$$

is generally not a KKT point for (1). However, since ϵ_g is a fairly arbitrary value to separate cluster representatives, the distinction between formulations (1) and (2) is minimal. For a relatively small number of clusters, say $k < 10$, it is possible to select a large κ , say $\kappa = 100$, without encountering numerical difficulties summing the $\binom{k}{2}$ terms in $Z(X)$. Thus, for a moderate number of cluster representatives, $Z(X)$ is a practical way to combine the g_i constraints, reducing the number of dual variables and hence the dimension of the homotopy map.

Positively Oriented NCP Functions

A continuous function $\hat{\Psi} : R \times R \rightarrow R$ is called an *NCP function* if $\hat{\Psi}(a, b) = 0 \iff 0 \leq a \perp b \geq 0$, and it is *positively oriented* if $\hat{\Psi}(a, b) \geq 0 \iff a \geq 0$ and $b \geq 0$. The positively oriented NCP function of interest here, first introduced by [13], is

$$\hat{\Phi}(a, b) = -|a - b|^3 + a^3 + b^3.$$

Observe that $\hat{\Phi}$ is C^2 ; moreover, for $b > 0$, $\hat{\Phi}(\cdot, b)$ is strictly increasing and onto \mathbb{R} . NCP functions are used to represent the complementarity conditions within the KKT necessary conditions: for each inequality constraint $g_i \leq 0$ with associated Lagrange multiplier μ_i , $\hat{\Phi}(-g_i, \mu_i) = 0 \iff -g_i \mu_i = 0$, $-g_i \geq 0$, $\mu_i \geq 0$. Thus, for an optimization problem (1) containing only inequality constraints, finding a KKT point is equivalent to solving the nonlinear system of equations

$$\begin{aligned} \nabla_X F(C, X) - \mu_0 \nabla \Psi(X) + \sum_{i=1}^{\binom{k}{2}} \mu_i \nabla g_i(X) &= 0, \\ \hat{\Phi}(\Psi, \mu_0) &= 0, \\ \hat{\Phi}(-g_i, \mu_i) &= 0, \quad i = 1, \dots, \binom{k}{2}. \end{aligned}$$

Homotopy Theory

Standard continuation methods ([17], [16]) find a root \bar{x} of a differentiable function $f(x)$ using a known root x_0 of a simple differentiable function $g(x)$ by solving

$$\rho(\lambda, x) = (1 - \lambda)g(x) + \lambda f(x) = 0$$

as λ is increased from 0 to 1, starting with the known solution x_0 at $\lambda = 0$. λ is the continuation parameter, g is called the ‘start’ function, and f is called the ‘target’ function. Given a solution (λ, x_λ) , standard local methods (such as Newton’s method) are used to solve $\rho(\lambda + \delta\lambda, x) = 0$ for fixed small $\delta\lambda > 0$. This yields a series of solutions along a zero curve γ of $\rho(\lambda, x)$. However, there is no guarantee that a given starting function g will yield a zero of f , as the algorithm may fail at some intermediate $\tilde{\lambda}$ as continuation progresses.

Continuation can fail if the zero curve γ of ρ emanating from $(0, x_0)$ fails to exist past some $\tilde{\lambda} < 1$. γ can just stop at $\tilde{\lambda}$,

turn back toward $\lambda = 0$ at $\tilde{\lambda}$, or go to infinity. γ may exist past $\tilde{\lambda}$, but bifurcate at $\tilde{\lambda}$, causing the local iteration to fail because $D_x \rho(\lambda, x)$ is singular at the bifurcation point $(\tilde{\lambda}, x_{\tilde{\lambda}})$.

Homotopy methods deal with bifurcation and turning points through a local parametrization of the zero curve $(\lambda, x) = (\lambda(t), x(t))$. Most importantly, homotopy methods treat λ as an independent variable, and do not increase λ monotonically from 0 to 1. The issues of nonexistence, bifurcation, and divergence to infinity are addressed by modern *probability-one homotopy* methods [17, 16, 4], which guarantee almost surely (in the probability measure theoretic sense) the existence of a smooth, nonbifurcating, bounded zero curve γ of a homotopy map $\rho_a(\lambda, x)$ that connects a start point $(0, x_0)$ to a point $(1, \bar{x})$, where $f(\bar{x}) = 0$.

These algorithms are implemented in FORTRAN 77 as HOMPACT [18], and extended in Fortran 90 as HOMPACT90 [19]. The following theorems about probability-one homotopy maps and the associated zero curves γ are central.

Theorem 1: Parametrized Sard's Theorem. Let $U \subset \mathbb{R}^m$, $V \subset \mathbb{R}^n$ be nonempty open sets, $\rho : U \times [0, 1) \times V \rightarrow \mathbb{R}^n$ be a C^2 map, and define

$$\rho_a(\lambda, x) = \rho(a, \lambda, x).$$

If ρ is transversal to zero (rank $D\rho = n$ on $\rho^{-1}(0)$), then for almost all $a \in U$ the map ρ_a is also transversal to zero.

Theorem 2. Let $F : \mathbb{R}^n \rightarrow \mathbb{R}^n$ and $\rho : \mathbb{R}^m \times [0, 1) \times \mathbb{R}^n \rightarrow \mathbb{R}^n$ be C^2 , and define $\rho_a(\lambda, x) = \rho(a, \lambda, x)$. Assume that

- (1) ρ is transversal to zero;
- (2) for each fixed $a \in \mathbb{R}^m$, $\rho_a(0, x) = 0$ has a unique solution x_a at which rank $D_x \rho_a(0, x_a) = n$;
- (3) $\rho_a(1, x) = F(x)$;
- (4) for each $a \in \mathbb{R}^m$, the connected component of the zero set $\rho_a^{-1}(0)$ containing $(0, x_a)$ is bounded.

Then for almost all $a \in \mathbb{R}^m$ there exists a zero curve γ of $\rho_a(\lambda, x)$, emanating from $(0, x_a)$, along which the $n \times (n+1)$ Jacobian matrix $D\rho_a(\lambda, x)$ has full rank, that does not intersect itself and is disjoint from any other zeros of ρ_a , and accumulates at a point $(1, \bar{x})$ for which $F(\bar{x}) = 0$. Furthermore, if rank $D\rho_a(1, \bar{x}) = n$, then the curve γ connecting $(0, x_a)$ to $(1, \bar{x})$ has finite arc length.

Theorem 3. Let $F : \mathbb{R}^n \rightarrow \mathbb{R}^n$ be C^2 , and suppose there exist $r_0, r > 0$, $r > r_0$, such that for any $a \in \mathbb{R}^n$ with $\|a\|_2 < r_0$, $x - a$ and $F(x)$ do not point in opposite directions on $\{x \in \mathbb{R}^n \mid \|x\|_2 = r\}$. Define $\rho : \mathbb{R}^n \times [0, 1) \times \mathbb{R}^n \rightarrow \mathbb{R}^n$ by

$$\rho(a, \lambda, x) = (1 - \lambda)(x - a) + \lambda F(x),$$

and let $\rho_a(\lambda, x) = \rho(a, \lambda, x)$. Then for almost all vectors $a \in \mathbb{R}^n$ with $\|a\|_2 < r_0$ there exists a zero curve γ of $\rho_a(\lambda, x)$, emanating from $(0, a)$, along which the $n \times (n+1)$ Jacobian matrix $D\rho_a(\lambda, x)$ has full rank, that does not intersect itself

and is disjoint from any other zeros of ρ_a , and accumulates at a point $(1, \bar{x})$ for which $F(\bar{x}) = 0$. Furthermore, if rank $D\rho_a(1, \bar{x}) = n$, then the curve γ connecting $(0, a)$ to $(1, \bar{x})$ has finite arc length.

Theorem 1, combined with Theorems 2 and 3, guarantee that the curve γ is the only curve emanating from $\lambda = 0$ and that γ must accumulate at $\lambda = 1$. Thus, a probability-one homotopy algorithm simply tracks the zero curve γ_a of ρ_a , which is guaranteed to reach a solution \bar{x} of $F(x) = 0$ at $\lambda = 1$, with probability one (almost surely) so long as the hypothesis of Theorems 2 and 3 are met. (Theorem 3 is simply a special case of Theorem 2.)

In practice, the full rank of the Jacobian matrix $D\rho_a(1, \bar{x})$ is not necessary, as the zero curve usually approaches a solution as $\lambda \rightarrow 1$ with finite arc length. This is especially true when applied to the semisupervised clustering problem, as the desired clustering (which satisfies the given constraints) is expected to be present at some point along γ before $\lambda = 1$. Homotopy maps that fulfill the theorems' assumptions are called *globally convergent probability-one homotopy maps*. Proving a map to be globally convergent with probability one reduces to proving that it meets the given assumptions. Given time to trace the finite arc length of the solution curve with a robust enough tracker, such curves will inevitably yield a useful solution. More valuable, tracing the curve yields the entire parametrized solution trace for analysis, generating a tradeoff curve that can be further analyzed.

It is possible to modify the given "natural" homotopy map $(1 - \lambda)g(x) + \lambda f(x)$ by manipulating the λ parameter to yield more useful maps. In particular, where $g(x)$ lacks a unique zero (such as the clustering problem case here), a unique zero x_0 at $\lambda = 0$ can be enforced by modifying the map to

$$\rho_a(\lambda, x) = (1 - \tanh(60\lambda))(x - x_0) + \tanh(60\lambda) [(1 - \lambda)g(x) + \lambda f(x)],$$

where \tanh is the hyperbolic tangent function. $\tanh(60\lambda) \approx 1$ for $\lambda > 0.3$, to within 64-bit machine accuracy. Thus $\rho_a(\lambda, x) = 0$ has a unique solution $x = x_0$ at $\lambda = 0$, but for $\lambda > 0.3$ the map looks essentially like $(1 - \lambda)g(x) + \lambda f(x)$. Semisupervised learning problems often have such "natural" start functions g with multiple zeros, making this a useful general trick in homotopy map generation. (The rigorous convergence theory for this map actually uses $\tanh(60\lambda/(1 - \lambda))$, which is computationally indistinguishable from $\tanh(60\lambda)$ for $\lambda > 0.3$.)

Clustering Application

Let $k^0 \in \mathbb{R}^{kd}$ be some (presumably poor) solution to the unsupervised clustering problem for k clusters in d dimensions, generated by a traditional clustering approach, such as the K-Means algorithm. For the present discussion, consider each cluster assignment to be a "hard" assignment, that is, each data point is assigned to a single cluster determined by its distances from the cluster representatives, not assigned a probability of belonging to each cluster based on those distances.

It is worth noting at this juncture that disjunctive and conjunctive combinations of constraints can be represented by the penalty functions F_m and F_c defined in Section 2, which are of particular value when ϵ - and δ -constraints are considered. ϵ - and δ -constraints act upon groups of instances. ϵ -constraints dictate that any data point in a cluster must have another data point in that cluster within ϵ distance, or be the only data point in the cluster. δ -constraints dictate that any datapoint in a cluster must be at least δ distance from every datapoint that resides in a different cluster. Both of these types of constraints can be represented as disjunctions and conjunctions of must-link constraints [7].

Let C^1 and C^2 be constraints (must-link, cannot-link, or combinatorial) and let F^1 and F^2 be the corresponding penalty functions. Then $C^3 = C^1 \vee C^2$ has the corresponding penalty function $F^3 = F^1 F^2$. Similarly, $C^4 = C^1 \wedge C^2$ has the corresponding penalty function $F^4 = F^1 + F^2$. Observe that $F^3 = 0$ if and only if C^3 is satisfied, and $F^4 = 0$ if and only if C^4 is satisfied. Finally, observe that any number of must-link and cannot-link constraints can thus be combined in conjunctive normal form by summing products of these penalty functions. As such, these penalty functions can easily be adapted to represent penalty functions for ϵ - and δ -constraints.

By Observation 4 in Section 2.1, if it is possible to satisfy all of the constraints defined by C , then there exists a vector of cluster representatives \mathcal{X} such that the penalty function $F(C, \mathcal{X}) = 0$. This vector of cluster representatives represents a global minimum point of the function F at which $\nabla_X F(C, \mathcal{X}) = 0$. This suggests the homotopy map (where $a = k^0$)

$$\tilde{\rho}_a(\lambda, X) = (1 - \lambda)(X - a) + \lambda (\nabla_X F(C, X))^T.$$

This homotopy map is appealing: when $\lambda = 0$, the solution is simply the solution k^0 to the unsupervised clustering problem. When $\lambda = 1$, the solution, if one exists, represents a local minimum point (or stationary point) of the penalty function, which is based on the violation of constraints. This is not to say that the solution generated will satisfy all the constraints if such a solution is possible, as it is fairly easy to construct a degenerate set of constraints so that there is a local solution close to $X = k^0$. However, in practice this has not proven to be a problem.

$\tilde{\rho}_a$ is a probability-one homotopy map, but while it satisfies conditions (1), (2), and (3) in Theorem 2, it fails to satisfy condition (4), bounded γ . Furthermore, there is a trivial solution to all constraints at $x^1 = x^2 = \dots = x^k$, where all cluster representatives are equal. Thus, modifications must be made to the above map to accommodate the constraints outlined earlier in Section 2.

HOMOTOPY MAPS AND CLUSTERING

K-Means Approximation

In order for the tradeoff curve to reflect an accurate picture of the differences between clustering based solely on the cluster hypothesis and clustering based on the satisfaction of cluster constraints, it is important that the start function $g(x)$ accurately represent the state of the clustering as determined by

the cluster hypothesis and that $f(x)$ accurately represents the state of the clustering as determined by the clustering constraints. The latter case is handled by the optimization problem given above, but the former case requires a clustering formulation that will work in the context of a homotopy map.

The traditional K-Means clustering algorithm is the most popular clustering algorithm based on the cluster hypothesis available. However, the K-Means function $K : \mathbb{R}^{kd} \rightarrow \mathbb{R}$ to be minimized,

$$K(X) = \sum_{i=1}^k \sum_{y \in S_i} \|y - x^i\|^2,$$

where S_i is the set of all points in cluster i , with representative x^i , is not C^2 , since cluster assignment is not differentiable. Ideally, besides being a fair approximation of the K-Means clustering algorithm, the approximation $\hat{K}(X) : \mathbb{R}^{kd} \rightarrow \mathbb{R}$ will have two additional qualities: It must be C^3 , and $\nabla_X \hat{K}(X) : \mathbb{R}^{kd} \rightarrow \mathbb{R}^{kd}$ must be bounded in the feasible region.

The most common such approximation [14] for a given data set \hat{Y} is

$$\hat{K}(X) = \sum_{i=1}^{|\hat{Y}|} \frac{k}{\sum_{j=1}^k \frac{1}{\|y^i - x^j\|^2}}.$$

This continuously differentiable approximation arises from posing the original K-Means clustering problem as a sum of products of the weight or probability $P_{i,j}(X)$ that a data point $y^i \in \hat{Y}$ belongs to a particular cluster represented by $x^j \in \hat{X}$, measured here as

$$P_{i,j} = \frac{1}{\|y^i - x^j\|^2} \frac{1}{\sum_{m=1}^k \frac{1}{\|y^i - x^m\|^2}},$$

and a measure $\tilde{D}_{i,j}$ of the distance that the data point resides from the cluster representative, taken here as $\tilde{D}_{i,j} = \|y^i - x^j\|^2$, a simple square of the Euclidean distance from x^j to y^i . Thus, points very close to their cluster representatives have high probability of belonging to that cluster, but bring a corresponding low value to the final optimization function, since such points are considered ideal. Summing the products yields the approximation $\hat{K}(X) = \sum_{i=1}^{|\hat{Y}|} \sum_{j=1}^k P_{i,j} \tilde{D}_{i,j}$ after

cancellation. Note, however, that this cancellation may yield overflow if $\|y^i - x^j\| \approx 0$, in which case the summand for that index i is taken as zero. These singularities are removable, and $\hat{K}(X)$ is an entire function (in each of the components of X , viewed as a complex vector). Minimizing $\hat{K}(X)$ thus yields an approximation of the minimum of the original K-Means function.

Homotopy Map

Generally inequality constraints are easier to deal with than equality constraints, so consider replacing the equality constraint $G(X) = 0$ used for the first homotopy map $\tilde{\rho}_a$ by the inequality constraint $Z(X) \leq 0$ discussed earlier. Keep the same bounding constraint $\Psi(X) \geq 0$. Using the same modified NCP function Φ as before, the equation

$$\Phi(\lambda, \mu, \Psi(X), h_0) = 0$$

for $h_0 > 0$, $\Psi(k^0) > 0$, $\Phi(0, \mu_0, \Psi(k^0), h_0) = 0$, and $0 \leq \lambda < 1$ forces $\Psi(X) > 0$ along the zero curve γ . Similarly the equation

$$\Phi(\lambda, \nu, -Z(X), h_1) = 0$$

for $h_1 = 0$, $Z(k^0) < 0$, $\Phi(0, \nu_0, -Z(k^0), h_1) = 0$, and $0 \leq \lambda < 1$ forces $Z(X) < 0$ along γ . When $\lambda = 1$, these two equations enforce the KKT conditions for the constraints $-\Psi(X) \leq 0$, $Z(X) \leq 0$ and their Lagrange multipliers μ, ν , respectively.

The Lagrangian function associated with (2) is $\tilde{L}(X, \mu, \nu) = F(C, X) - \mu\Psi(X) + \nu Z(X)$, and a KKT point $(\bar{X}, \bar{\mu}, \bar{\nu})$ for (2) satisfies

$$\begin{aligned} \nabla_X \tilde{L}(X, \mu, \nu) &= 0, \\ 0 &\leq \mu \perp \Psi(X) \geq 0, \\ 0 &\leq \nu \perp -Z(X) \geq 0. \end{aligned}$$

Furthermore, should $Z(\bar{X}) < 0$, the KKT point $(\bar{X}, \bar{\mu}, 0)$ for (2) yields a KKT point $(\bar{X}, \bar{\mu}, 0, \dots, 0)$ for (1).

Finally, putting all the pieces together, the proposed constrained clustering homotopy map is $\tilde{\rho}_a(\lambda, X, \mu, \nu) = \begin{pmatrix} (1 - t(\lambda))(X - k^0) + t(\lambda)\varphi(\lambda, X, \mu, \nu) \\ \Phi(\lambda, \mu, \Psi(X), h_0) \\ \Phi(\lambda, \nu, -Z(X), h_1) \end{pmatrix}$, where $\varphi(\lambda, X, \mu, \nu) =$

$$((1 - \lambda)\nabla_X \hat{K}(X) + \lambda\nabla_X \tilde{L}(X, \mu, \nu))^T,$$

$a = (k^0, h_0, h_1)$, $t(\lambda) = \tanh(60\lambda)$ and k^0 is any point for which $\Psi(k^0) > 0$, $Z(k^0) < 0$, and $\nabla_X \hat{K}(k^0) \approx 0$, e.g., a K-Means solution (locally) minimizing $K(X)$.

The \tanh terms in the above construction arise because $\nabla_X \hat{K}(X) = 0$ has multiple possible solutions. At the very least, permutations of the cluster representatives in X will yield identical values for $\nabla_X \hat{K}$. The \tanh terms ensure that $\tilde{\rho}_a(0, X, \mu, \nu) = 0$ has a unique solution as required by Theorem 2. Since $h_0 > 0$, $\Phi(0, \mu, \Psi(k^0), h_0) = 0$ uniquely determines $\mu = \mu_0 > 0$, and similarly $h_1 > 0$, $\Phi(0, \nu, -Z(k^0), h_1) = 0$ uniquely determines $\nu = \nu_0 > 0$.

Computationally, as mentioned earlier,

$$\tanh(60\lambda) = 1$$

in 64-bit arithmetic for $\lambda > 0.3$, and thus, for $\lambda > 0.3$, this map functions identically to

$$\begin{pmatrix} ((1 - \lambda)\nabla_X \hat{K}(X) + \lambda\nabla_X \tilde{L}(X, \mu, \nu))^T \\ \Phi(\lambda, \mu, \Psi(X), h_0) \\ \Phi(\lambda, \nu, -Z(X), h_1) \end{pmatrix}.$$

EXPERIMENTAL RESULTS

Experiments to discover the effectiveness of the homotopy tracking algorithm with the proposed homotopy map, as compared to popular existing constrained clustering algorithms, are presented here. The constraints used involve combinations of ML and MNL constraints (solving problems involving solely ML constraints are fairly straightforward polynomial time graph problems).

The existence of MNL constraints in the constraint sets is crucial to understanding the complexity of the test problems. Davidson et al. [8] state that as a rough rule of thumb a set of constraints can be understood as fundamentally “difficult” for these iterative K-Means approaches if any single datapoint appears in k or more MNL constraints. As such, for each dataset presented here, both an “easy” and a “difficult” set of constraints were generated. The “easy” constraint set involves one hundred constraints such that no datapoint appears more than $k - 1$ times in a mix of ML and MNL constraints. The “difficult” constraint set, also one hundred constraints, involves predominately MNL constraints, and guarantees that at least one datapoint is involved in k MNL constraints. In both cases, the generated constraints were completely random, with no a priori knowledge about how well the generated constraints would guide the algorithms to a correct solution.

The datasets involved are all taken from the UCI machine learning dataset repository [1]. They represent a balanced selection of moderately easy clustering problems without constraints, and should demonstrate some of the key differences between the homotopy algorithms utilizing the maps $\tilde{\rho}$ and $\tilde{\rho}$ developed here and the K-Means algorithms used previously. The datasets are “Liver Disorders” (liver), “Pima Indians Diabetes” (pima), “Steel Plates Faults” (faults), “Wine” (wine), “Iris” (iris), “Ionosphere” (iono), “Glass Identification” (glass), and “PAMAP2 Physical Activity Monitoring” (pamap). The datasets “faults” and “pamap” were modified in the following manner: The first three classification categories of the dataset “faults” were treated as additional data, and the last classification category was used for classification. The dataset “pamap” was modified by eliminating all data points of the “0” classification (as recommended by the contributors) and any data point with a “NaN” data value. See Table 1 for the relevant details for each dataset.

The K-Means algorithms used for the comparison are those presented by [3]: metric pairwise constrained K-Means (MPCK-Means), metric learning K-Means without pairwise constraints (MK-Means), and pairwise constrained K-Means without metric learning (PCK-Means). The standard K-Means result is also presented, to be used as a baseline. These algorithms were chosen for several reasons. First, K-Means is by far the most popular clustering algorithm, if only because of its intuitive approach and ease of programming; thus, K-Means algorithms modified for constrained clustering are the most likely to be consulted by researchers who are interested in constrained clustering problems. Second, these constrained K-Means algorithms minimize a summed penalty function based on the distance from the cluster cen-

TABLE 1
Dataset Summary

	No. Instances	No. Features	No. Categories
liver	345	6	2
pima	768	7	2
faults	1941	31	2
wine	178	12	3
iris	150	4	3
iono	351	34	2
glass	214	10	6
pamap	175498	53	12

troids to the data points assigned to that cluster. While this penalty function may be discrete, it is still similar enough to the penalty function presented here to make comparisons between these algorithms and the homotopy approach feasible.

It is worth noting immediately that three things set the homotopy algorithms apart from the K-Means algorithms presented here. First, for the K-Means algorithms, the ordering of the constraints plays a nonnegligible role in the quality of the final result, meaning that finding the best result theoretically involves searching every permutation of a given constraint set (which is not computationally feasible). For a homotopy algorithm, the ordering of the constraints is unimportant. Second, not only are problems involving concentrations of MNL constraints involving the same datapoint not qualitatively more “difficult” for homotopy algorithms, but, since distances only need to be calculated once per iteration, problems involving concentrations of datapoints are computationally less intense than problems where the constraints are more diverse, at least until each datapoint is involved in at least one constraint. Finally, the homotopy algorithms, like the K-Means algorithm, is limited to convex clusterings, which for some datasets can be potentially debilitating. In contrast, the adapted K-Means algorithms presented here distinguish between cluster assignment and cluster centroids, which allows for nonconvex clusterings.

One reasonable question that arises in semisupervised learning is what kinds of information would need to be present in a clustering problem that could not be represented by the datapoints themselves. One answer makes reference to the cluster hypothesis itself.

The cluster hypothesis states that if two datapoints are close to each other (for a vague notion of closeness), then they should belong to the same cluster; if they are far apart, they should belong to different clusters. The constraints that formalize this statement are known as ϵ - and δ -constraints; ϵ -constraints are constraints that dictate that any data point in a cluster must have another data point in that cluster within ϵ distance, or represent the entire cluster, while δ -constraints state that any datapoint in a given cluster must be at least δ distance from every datapoint in a different cluster. Both of

these constraints can be represented as disjunctions and conjunctions of the classic “must-link” and “must-not-link” constraints [8]. The number of such constraints can grow quite large as the number of datapoints in the set increases (depending on the values assigned to ϵ and δ), but the entire set of constraints need not be brought to bear for the solution to show improvement. One advantage of such constraints is that they don’t depend on the “real” clustering, which is to say the classification, of a given dataset, which is often unknown in practice. Thus, applying these constraints to test problems can yield tests of improvement in whatever measure of cluster hypothesis satisfaction is desired (of which there are many).

100 random constraints were generated using ϵ - and δ -constraints based on reasonable values for the given data sets. Since the adjusted Rand index is useless in this context, the Davies-Bouldin index (DBI) was used instead. The DBI is a nonnegative measure of conformity to the cluster hypothesis; a lower DBI indicates closer conformity to the clustering hypothesis. The “pamap” dataset was not used due to the difficulty in generating meaningful differences in the clusterings with this sort of constraint. Table 2 shows these results. In these experiments the best cluster found by the homotopy algorithm was also uniformly the last one.

TABLE 2
Davies-Bouldin index, ϵ - and δ -constraints

	K	MK	PCK	MPCK	$\bar{\rho}$
liver	1.7349	2.3067	1.7679	1.2516	0.8706
	1.7349	1.8801	1.4568	1.2516	0.8706*
	1.7349	1.6682	1.3542	1.2516	0.8706
pima	1.9995	0.9883	0.8762	0.8681	0.8094
	1.5653	1.9403	1.0585	1.4436	0.8601*
	1.5387	1.9316	1.0585	1.4436	0.8601
faults	0.9392	0.9883	0.8762	0.8681	0.8094
	0.9392	0.9652	0.8762	0.8681	0.8094*
	0.9392	0.9652	0.8637	0.8681	0.8094
wine	1.5126	1.6650	0.8185	1.5393	0.6604
	1.5126	1.5507	0.6542	1.4515	0.6097*
	1.5126	1.4506	0.6101	1.3447	0.4948
iris	0.7373	1.5023	1.4662	0.9612	0.9379
	0.7373	0.9455	0.8877	0.7175	0.6453*
	0.7373	0.7445	0.7041	0.6585	0.5776
iono	2.0706	2.0512	1.6898	1.6898	1.6188
	2.0706	1.8936	1.8936	1.6898	1.6188*
	2.0706	1.8936	1.8919	1.6898	1.6188
glass	3.4599	1.8348	1.0414	1.8284	2.2789
	2.2910	1.4204	1.0414*	1.2820	1.2204
	1.7415	1.0621	1.0414	1.0038	0.2403

5. DISCUSSION

The adjusted Rand index is a good tool for a posteriori judgment of clusters [10], but semisupervised clustering problems don’t have the classification a priori. The tools of the researcher are (usually) limited to intercluster and intracluster distances, with limited extra information not presented as a dimension of the clustering. For this sort of situation, the

homotopy method shines in the ϵ - and δ -constraint experiments (Table 2), due to the use of the K-Means approximation $\hat{K}(X)$ in the homotopy formulation. The net effect of this approximation is to cause the homotopy method to account for local minima of the K-Means approximation as λ increases. Assuming that $\Psi(\hat{X}) > 0$ and $Z(\hat{X}) < 0$, with a small ν and μ , which is almost always the case in practice, an \hat{X} at some $\hat{\lambda} < 1$ that would satisfy $\nabla_X F(C, \hat{X}) \approx 0$, but violate $\nabla_X \hat{K}(\hat{X}) \approx 0$, would not lie along γ . Thus, γ contains those solutions that do not strongly violate the clustering hypothesis as arc length increases along γ , resulting in the end point at $\lambda = 1$ being generally favorable to the cluster hypothesis.

The new homotopy approach for constrained clustering problems uses state-of-the-art mathematical software to characterize multicriteria problems in constrained clustering. Just as in other applications of homotopy methods to science and engineering, the application of homotopy methods to machine learning problems can usher in greater understanding of solution sets and the value of constraints. Besides the strong mathematical foundations and rigorous formalisms brought to classical machine learning problems, this homotopy approach has the potential to greatly reduce the ad hoc nature of methodological experimentation that is prevalent in practice. The approach given here not only helps extract better patterns from data, but also helps formally understand the internal workings of machine learning techniques. Future work includes homotopy maps for other multicriteria machine learning problems such as information bottleneck, time series segmentation, and transfer learning.

REFERENCES

1. Bache, K., and Lichman, M. Uci machine learning repository, 2013. <http://archive.ics.uci.edu/ml>.
2. Basu, S., Davidson, I., and Wagstaff, K. *Constrained Clustering: Advances in Algorithms, Theory, and Applications*. Chapman and Hall, 2008.
3. Bilenko, M., Basu, S., and Mooney, R. J. Integrating constraints and metric learning in semi-supervised clustering. In *ICML '04* (2004), 11–18.
4. Chow, S. N., Mallet-Paret, J., and Yorke, J. A. Finding zeros of maps: homotopy methods that are constructive with probability-one. *Math. Comput.* 32 (1978), 887–899.
5. Corduneanu, A., and Jaakkola, T. Continuation methods for mixing heterogeneous sources. In *UAI '02* (2002), 111–118.
6. Davidson, I. Two approaches to understanding when constraints help clustering. In *KDD '12* (2012), 1312–1320.
7. Davidson, I., and Ravi, S. Clustering with constraints: feasibility issues and the k-means algorithm. In *SDM '05* (2005), 201–211.
8. Davidson, I., and Ravi, S. Identifying and generating easy sets of constraints for clustering. In *AAAI '06* (2006), 336–341.
9. Davidson, I., and Ravi, S. S. The complexity of non-hierarchical clustering with instance and cluster level constraints. *Data Min. Knowl. Discov.* 14 (2007), 25–61.
10. Easterling, D. R., Hossain, M. S., Watson, L. T., and Ramakrishnan, N. Probability-one homotopy maps for tracking constrained clustering solutions. In *Proc. 2013 Spring Simulation Multiconference, HPC Symp., F. Liu, K. Rupp, R. Phillips and W. I. Thacker (eds.)*, Soc. for Modelling and Simulation Internat. (2013), 142–149.
11. Ji, S., Watson, L. T., and Carin, L. Semisupervised learning of hidden Markov models via a homotopy method. *IEEE Trans. Pattern Anal. Machine Intell.* 31 (2009), 275–287.
12. Kreisselmeier, G., and Steinhauser, R. Systematic control design by optimizing a vector performance index. In *Internat. Fed. of Active Controls Symp. on Computer-Aided Design of Control Systems* (1979).
13. Mangasarian, O. Equivalence of the complementarity problem to a system of nonlinear equations. *SIAM Journal on Applied Mathematics* 31, 1 (1976), 88–92.
14. Phillips, R. A Probabilistic Classification Algorithm with Soft Classification Output, 2009.
15. Sinha, K., and Belkin, M. The value of labeled and unlabeled examples when the model is imperfect. *Advances in Neural Information Processing Systems* 20 (2008).
16. Watson, L. T. Probability-one homotopies in computational science. *J. Comput. Appl. Math.* 140 (2002), 785–807.
17. Watson, L. T. A globally convergent algorithm for computing fixed points of C^2 maps. *Appl. Math. Comput.* 5 (2008), 297–311.
18. Watson, L. T., Billups, S., and Morgan, A. Algorithm 652: HOMPACk: a suite of codes for globally convergent homotopy algorithms. *ACM Trans. Math. Software* 13 (1987), 281–310.
19. Watson, L. T., Sosonkina, M., Melville, R. C., Morgan, A., and Walker, H. Algorithm 777: HOMPACk90: a suite of Fortran 90 codes for globally convergent homotopy algorithms. *ACM Trans. Math. Software* 23 (1997), 514–549.
20. Yang, H., and Callan, J. A metric-based framework for automatic taxonomy induction. In *ACL '09* (2009), 271–279.
21. Zhang, D., He, J., Liu, Y., Si, L., and Lawrence, R. Multi-view transfer learning with a large margin approach. In *KDD '11* (2011), 1208–1216.

Productive Parallel Programming with CHARM++

Phil Miller
Department of Computer Science
University of Illinois
mille121@illinois.edu

Keywords: Parallel programming, productivity, load balancing, overdecomposition

1. INTRODUCTION

CHARM++ is a general-purpose framework for developing high-performance parallel applications [1]. Applications written using CHARM++ run at scales spanning mobile devices [2], multi-core processors, multi-processor NUMA workstations and servers, networked clusters, and the world's largest supercomputers. A selection of CHARM++ applications in production scientific usage is shown in Table 1. These applications consume the second most execution cycles on NSF computing resources, after MPI.

The key goal of CHARM++ is high scientific productivity. This takes account of the notion that the time to solve a scientific problem with computation includes both development time and execution time. At present, a substantial portion of development time on large-scale parallel applications is directed toward performance optimization and implementing features that are necessary in many such applications, such as load balancing. Because of limitations on developer time and differences in expertise, application-specific renditions of these features tend to be fairly rudimentary.

CHARM++ addresses this challenge through robust *separation of concerns*. Application developers and scientific users are expected to apply their domain knowledge to define scalable parallel algorithms that suit their purpose. The CHARM++ runtime system takes responsibility for ensuring those algorithms execute efficiently, across the full range of platforms. Where necessary, CHARM++ provides plugin interfaces for highly application-dependent tasks (such as mapping and load balancing) while ensuring that application logic is unaffected by changes in these areas.

In light of its widespread use, CHARM++ is maintained as a high-quality software project suitable for production, rather than just CS research. Bugs in the system are tracked and aggressively fixed. Changes are verified through a bundled test suite and continuous integration with several applications. They are also reviewed by a developer other than their author for problems before integration.

2. TUTORIAL

This short tutorial aims to get new CHARM++ users over the initial hurdles, to enable subsequent independent exploration and development. These hurdles include both conceptual issues, like the elements of a CHARM++ program and design philosophy, as well as mechanical issues like compilation and language syntax. With that foundation, we will discuss the design of a few example applications and usage of key features of the CHARM++ system.

2.1. Download, build, install CHARM++

CHARM++ is distributed in source form on the web at <http://charmplusplus.org/>. Once downloaded, users need merely run `./build` to activate an interactive script that will generate and compile a suitable configuration.

2.2. Design Philosophy of CHARM++ Applications: Overdecomposition

The execution model of CHARM++ is that every processor in the parallel system is assigned a set of objects whose work will be executed on that processor. These objects are activated by the delivery of asynchronous messages through their host processor's scheduling queue. Good CHARM++ applications typically present many objects per processor, an approach that we term *overdecomposition* in contrast to the process-oriented decomposition typical of MPI and PGAS models. These objects are generally representative of units in the problem domain (e.g. matrix blocks, chunks of an interaction pattern, or blocks of a data-parallel structure).

This structure yields several benefits. The first benefit is processor count independence. Since the objects are defined by the input problem and program configuration, and mapped to processors by the runtime system, programs can run on any number of processors. This is a major natural convenience over older-style MPI applications that were often constrained to run on processor counts that were a specific function of the input size, like a power of 2 or a cube.

The next benefit is adaptive, asynchronous execution. With multiple objects on each processor, only one must be ready to execute at a time to ensure that they don't run out of work and go idle. The runtime system can fully overlap communication to and from each object with computation on others. The relative priority of different units of work can also be adapted dynamically, either based on application hints or runtime instrumentation and control mechanisms.

Table 1. A selection of applications built on CHARM++. Scaling of “> 350k cores” indicates effective use of the full Blue Waters Cray XE6 system at NCSA. SLOC are measured using David A Wheeler’s ‘SLOCcount’

Application	Domain	SLOC	Scalability (cores)
NAMD	Molecular biophysics	118k	> 350k
ChaNGa	Cosmology & Astrophysics	74k	> 350k
EpiSimdemics	Agent-base epidemiology	N/A	> 350k
OpenAtom	Material science	97k	40k
Enzo-P/Cello	Cosmology & Astrophysics	52k	32k
FreeON/SpAMM	Quantum Chemistry	111k	24k
ROSS	Discrete Event Simulation	N/A	16k
SDG	Engineering mechanics	N/A	10k
ClothSim	Cloth Physics with Rigid Bodies	N/A	768
JetAlloc	Stochastic mixed-integer program optimization	N/A	120

Given overdecomposition and the *migratability* of the objects, the runtime system is further empowered to provide efficient execution. Based on measured or predicted work required on each object, the mapping of objects to processor can be adjusted dynamically to provide *load balance*. CHARM++ provides both the instrumentation and migration infrastructure to enable this, and a suite of plug-in strategies to generate new mappings. Various strategies consider a wide range of factors beyond simply load, such as communication locality, processor speed, and interference from other processes and co-located VMs on a common host.

With the serialization mechanisms necessary to migrate objects for load balancing purposes, CHARM++ is also able to take snapshots of those objects for *fault tolerance* purposes. At its simplest, this consists of an application-agnostic pure userspace for checkpointing to files on stable storage and restarting from those files later. The system can restart applications on a different number of nodes and processes than were used in the run that generated a checkpoint, granting substantial flexibility. In supportive environments, CHARM++ also provides more sophisticated mechanisms that implement online fault tolerance, that allow applications to continue running through the loss of one or even several nodes and the occurrence of data corruption. These include in-memory checkpoints, message logging, and cross-checked replicated execution.

2.3. MPI Interoperation

Code written in CHARM++ and MPI can interoperate within a single parallel job [3]. This provides application developers with substantial flexibility. It means that individual modules of an application can be written in one model or the other as appropriate. In particular, it allows developers writing primarily in one model to take advantage of pre-existing libraries written in the other. It also enables incremental migration between the two different environments, converting

individual components as the need arises rather than porting an entire codebase in a single step.

REFERENCES

- [1] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, and L. Kale, “Parallel Programming with Migratable Objects: Charm++ in Practice,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC ’14. New York, NY, USA: ACM, 2014.
- [2] Touchpress Limited. *Molecules by Theodore Gray*. [Online]. Available: <https://itunes.apple.com/us/app/molecules-by-theodore-gray/id923383841?mt=8>
- [3] N. Jain, A. Bhatele, J.-S. Yeom, M. F. Adams, F. Miniati, C. Mei, and L. V. Kale, “Charm++ & MPI: Combining the best of both worlds,” in *Proceedings of the IEEE International Parallel & Distributed Processing Symposium (to appear)*, ser. IPDPS ’15. IEEE Computer Society, May 2015, ILNL-CONF-663041.

Biography

Phil Miller is a PhD candidate in the Department of Computer Science at the University of Illinois at Urbana-Champaign, focusing on the development of asynchronous parallel algorithms. He is a member of the Parallel Programming Laboratory, directed by Professor Laxmikant (Sanjay) Kalé. Phil has been a core developer of CHARM++ for several years, and managed feature and patch releases spanning versions 6.3.0 through 6.6.0. He has given presentations and tutorials on CHARM++ at numerous conferences and national laboratories.

This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (awards OCI-0725070 and ACI-1238993) and the state of Illinois.